US006199204B1

(12) United States Patent
Donohue

(10) **Patent No.: US 6,199,204 B1**
(45) **Date of Patent: Mar. 6, 2001**

(54) **DISTRIBUTION OF SOFTWARE UPDATES VIA A COMPUTER NETWORK**

(75) Inventor: **Seamus Donohue**, Artane (IR)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/158,704**

(22) Filed: **Sep. 22, 1998**

(30) **Foreign Application Priority Data**

Jan. 28, 1998 (GB) 9801661

(51) **Int. Cl.**^{7} **G06F 9/445**

(52) **U.S. Cl.** **717/11**; 705/59

(58) **Field of Search** 717/3, 11; 709/203, 709/216–223; 707/200, 203; 705/26, 27, 57, 58, 59; 713/191, 189

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,558,413		12/1985	Schmidt et al.	364/300
5,155,847		10/1992	Kirouac et al.	395/600
5,473,772	*	12/1995	Halliwell et al.	395/712
5,581,764		12/1996	Fitzgerald et al.	395/703
5,752,042	*	5/1998	Cole et al.	395/712
5,778,231		7/1998	Hoff et al.	395/705
5,797,016		8/1998	Chen et al.	395/712
5,809,251	*	9/1998	May et al.	709/223
5,845,090	*	12/1998	Collins, III et al.	709/221
5,845,293		12/1998	Veghte et al.	707/202
5,859,969	*	1/1999	Oki et al.	395/200.3
5,867,714	*	2/1999	Todd et al.	395/712
5,870,610	*	2/1999	Beyda	395/712
5,881,236	*	3/1999	Dickey	709/221
5,933,647		8/1999	Aronberg et al.	395/712
5,983,241		11/1999	Hoshino	707/203
5,991,771		11/1999	Falls et al.	707/202
5,999,740	*	12/1999	Rowley	395/712
5,999,947		12/1999	Zollinger et al.	707/203
6,006,034		12/1999	Heath et al.	395/712

FOREIGN PATENT DOCUMENTS

2 321 322	7/1998	(GB) .
WO 92/22870	12/1992	(WO) .
WO 94/25923	11/1994	(WO) .
WO98/07085	2/1998	(WO) .

OTHER PUBLICATIONS

Marimba Products "Castanet" from http://www.marimba.com/products website, p. 1.
Novadigm Fact Sheet, Novadigm's EDM from http://www/novadigm.com/cb5.htm website pp. 1 and 2, and http://www.novadigm.com/cb2.htm website, pp. 1 and 2.
"Internet Finalist: Castanet", Castanet Development Team, from http://www.zdnet.com/...ts/content/pcmg/1622/pcmg0079.html website, 1997 p. 1.

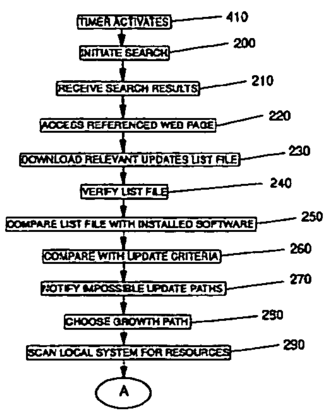
(List continued on next page.)

Primary Examiner—Kakali Chaki
(74) *Attorney, Agent, or Firm*—Jeanine S. Ray-Yarletts

(57) **ABSTRACT**

Provided is a method and mechanism for automating updating of computer programs. Conventionally, computer programs have been distributed on a recording medium for users to install on their computer systems. Each time fixes, additions and new versions for the programs were developed, a new CD or diskette was required to be delivered to users to enable them to install the update. More recently some software has been downloadable across a network, but the effort for users obtaining and installing updates and the effort for software vendors to distribute updates remains undesirable. The invention provides an updater agent which is associated with a computer program and which accesses relevant network locations and automatically downloads and installs any available updates to its associated program if those updates satisfy predefined update criteria of the updater agent. The updater agents are able to communicate with each other and so a first updater agent can request updates to programs which are prerequisites to its associated program.

9 Claims, 5 Drawing Sheets



TIMER ACTIVATES
410
INITIATE SEARCH
200
RECEIVE SEARCH RESULTS
210
ACCESS REFERENCED WEB PAGE
220
DOWNLOAD RELEVANT UPDATES LIST FILE
230
VERIFY LIST FILE
240
COMPARE LIST FILE WITH INSTALLED SOFTWARE
250
COMPARE WITH UPDATE CRITERIA
260
NOTIFY IMPOSSIBLE UPDATE PATHS
270
CHOOSE GROWTH PATH
280
SCAN LOCAL SYSTEM FOR RESOURCES
290
A

OTHER PUBLICATIONS

"Review: Marimba Castanet 3.0", Bradley F Shimmon, from http://www/lantimes.com/testing/98aug/808c011a.html website, 1998, pp. 1–4.

Pell et al "Mission Operations with and Automous Agent" Aerospace Conference, 1998 IEEE, Mar. 21–28, 1998, vol. 2, pp. 289–313.

Yeung et al "A Multi–Agent Based Tourism Kiosk on Internet" Proceedings of the Thirty–First Hawaii Int'l Conf on System Sciences, Jan 6–8, 1998, vol. 4, pp. 452–461.

Computer Database record 02071725 of Home Office Computing, v15, n5, p66(5), May 1997, D Haskin, "Save time while you sleep (round–the–clock computing chores)".

Computer Database record 02046777 of Computer Shopper, v16, n4, p568(5), Apr. 1997, D Aubrey, "Changing channels (Internet broadcasting)".

Computer Database record 02046764 of Computer Shopper, v16, n4, p540(2), Apr. 1997, L Bailes, "First Aid 97: a good does of preventive medicine (Cybermedia First Aid 97)".

Automating Internet Service Management, "The First Distributed Object–Oriented Automation Environment for Managing Internet Services", James Herman, Northeast Consulting Resources Inc.

* cited by examiner

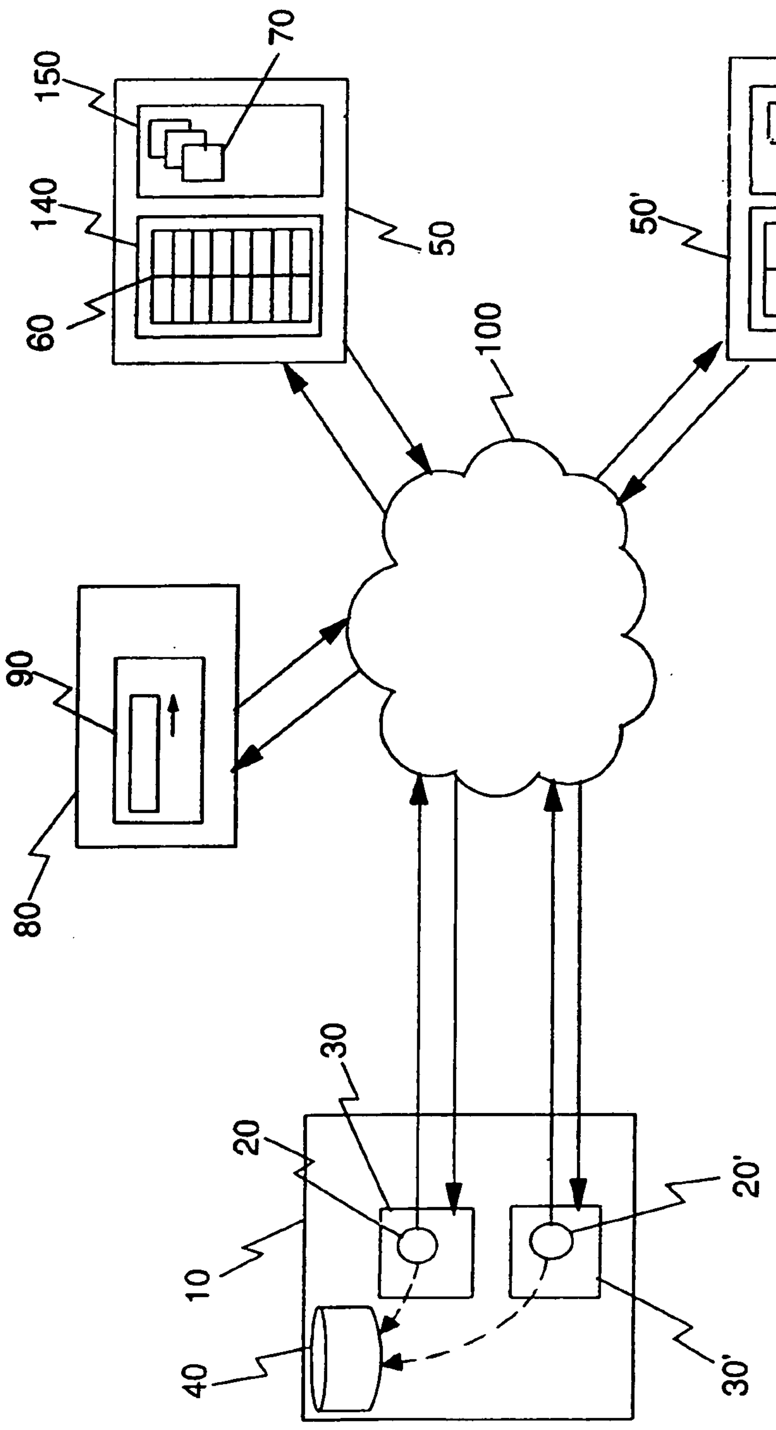
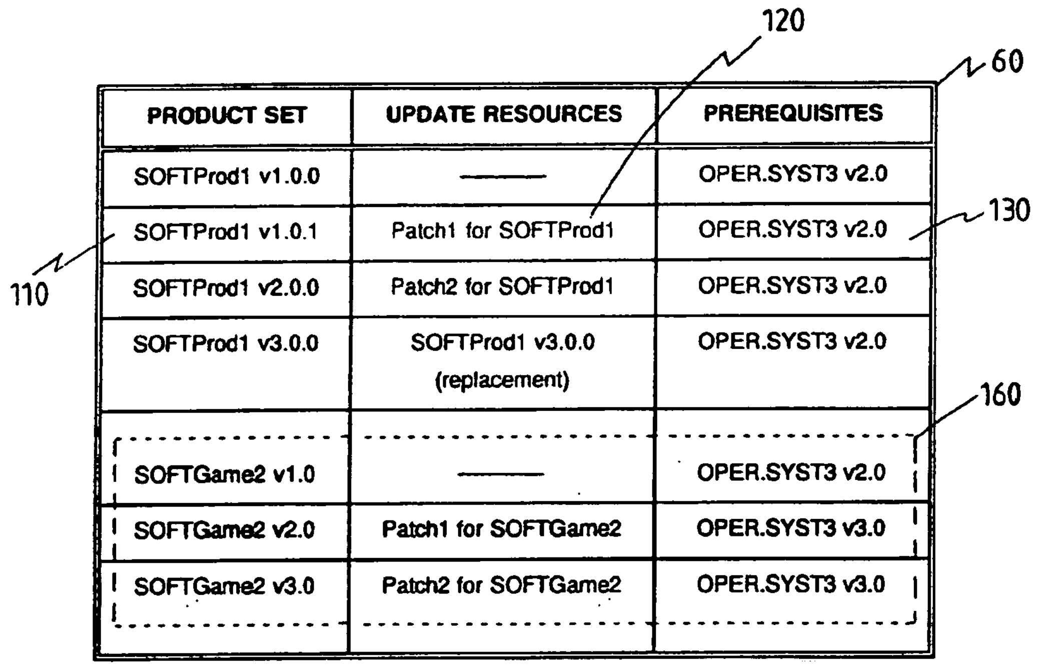
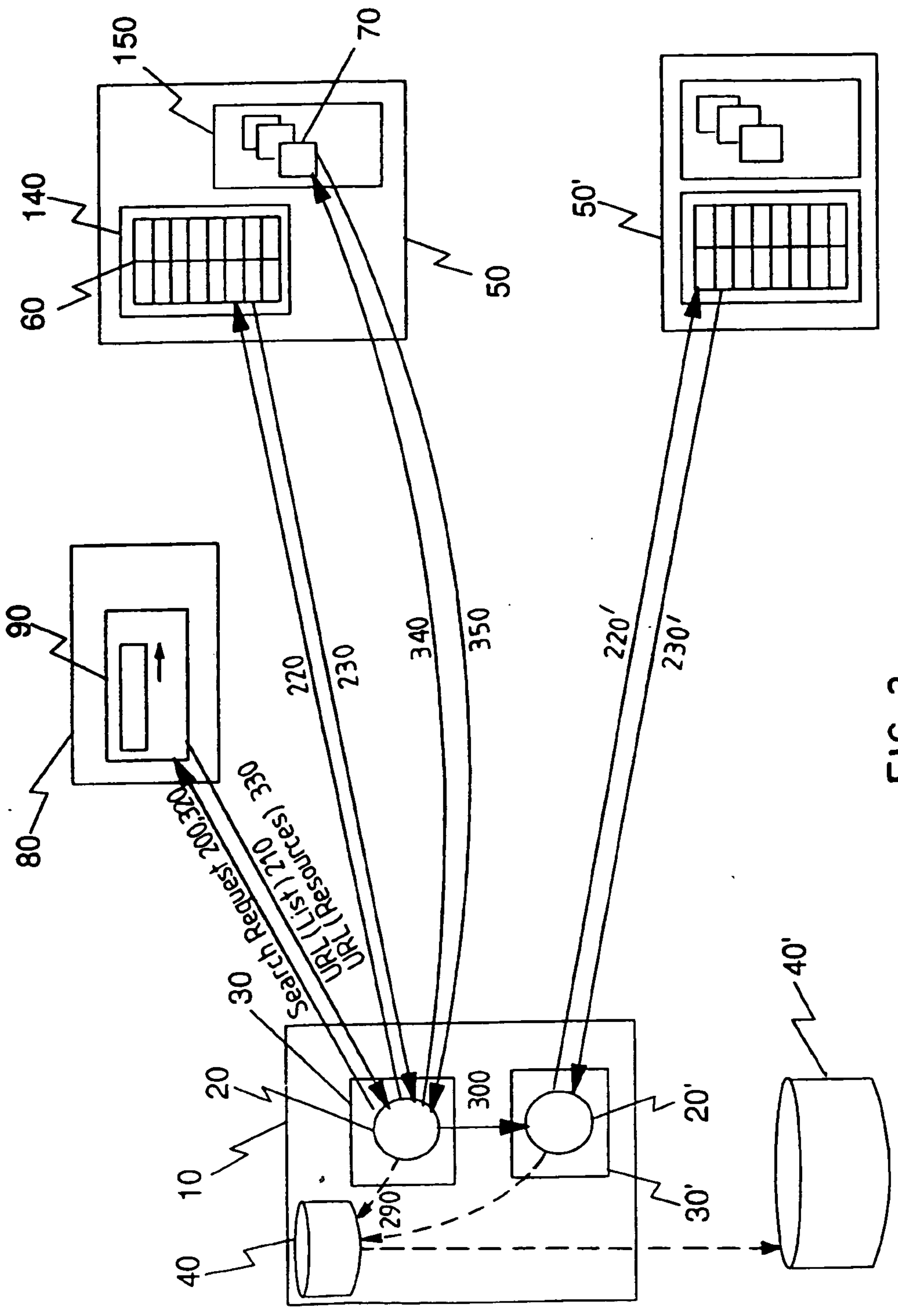


FIG. 1


120
60
130
110
160

PRODUCT SET	UPDATE RESOURCES	PREREQUISITES
SOFTProd1 v1.0.0	———	OPER.SYST3 v2.0
SOFTProd1 v1.0.1	Patch1 for SOFTProd1	OPER.SYST3 v2.0
SOFTProd1 v2.0.0	Patch2 for SOFTProd1	OPER.SYST3 v2.0
SOFTProd1 v3.0.0	SOFTProd1 v3.0.0 (replacement)	OPER.SYST3 v2.0
SOFTGame2 v1.0	———	OPER.SYST3 v2.0
SOFTGame2 v2.0	Patch1 for SOFTGame2	OPER.SYST3 v3.0
SOFTGame2 v3.0	Patch2 for SOFTGame2	OPER.SYST3 v3.0

FIG. 2



150
70
140
60
50
50'
90
80
220
230
340
350
220'
230'
Search Request 200,320
URL (List) 210
URL (Resources) 330
30
20
300
10
20'
290
40
30'
40'

FIG. 3

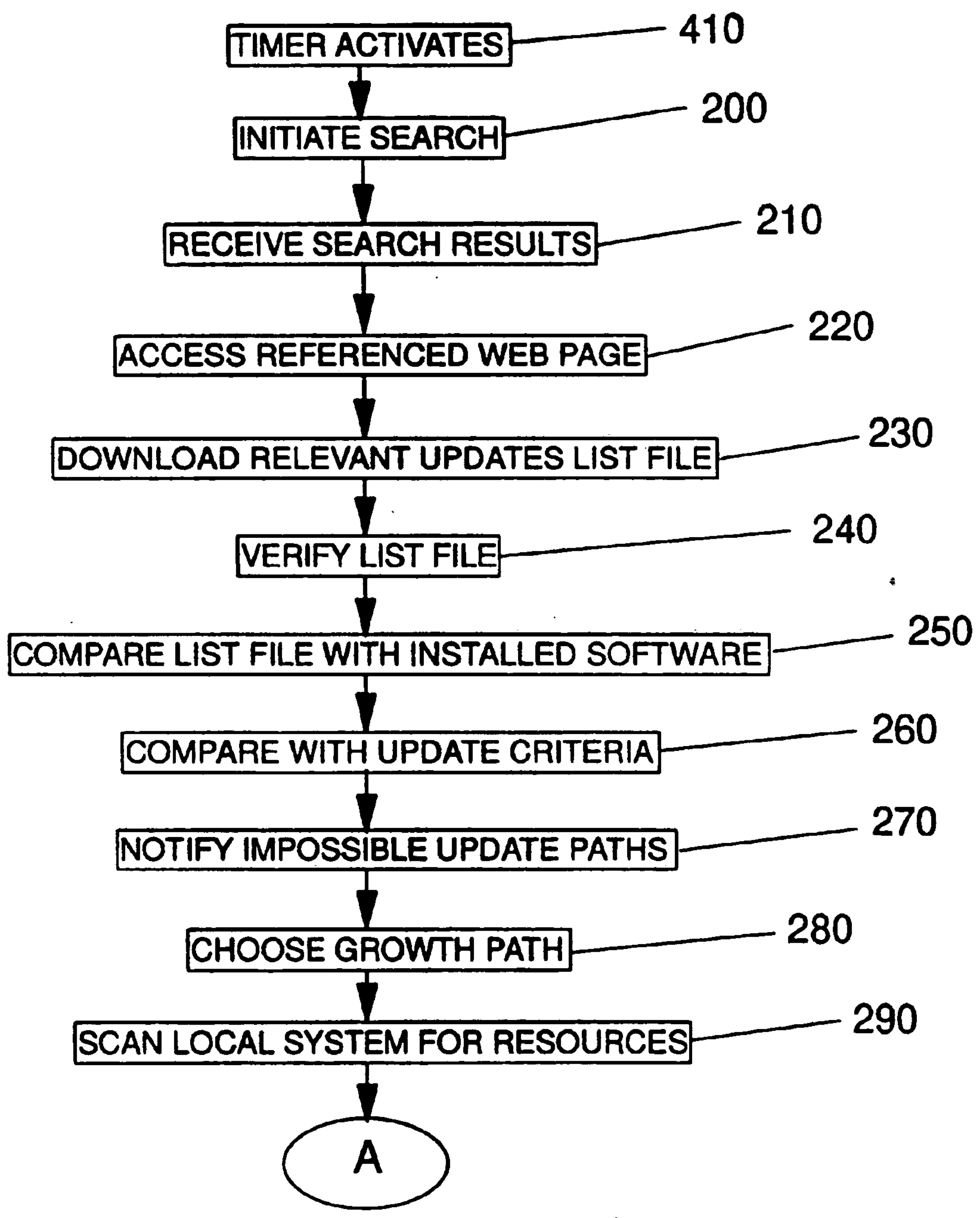

TIMER ACTIVATES
410
INITIATE SEARCH
200
RECEIVE SEARCH RESULTS
210
ACCESS REFERENCED WEB PAGE
220
DOWNLOAD RELEVANT UPDATES LIST FILE
230
VERIFY LIST FILE
240
COMPARE LIST FILE WITH INSTALLED SOFTWARE
250
COMPARE WITH UPDATE CRITERIA
260
NOTIFY IMPOSSIBLE UPDATE PATHS
270
CHOOSE GROWTH PATH
280
SCAN LOCAL SYSTEM FOR RESOURCES
290
A

FIG. 4A

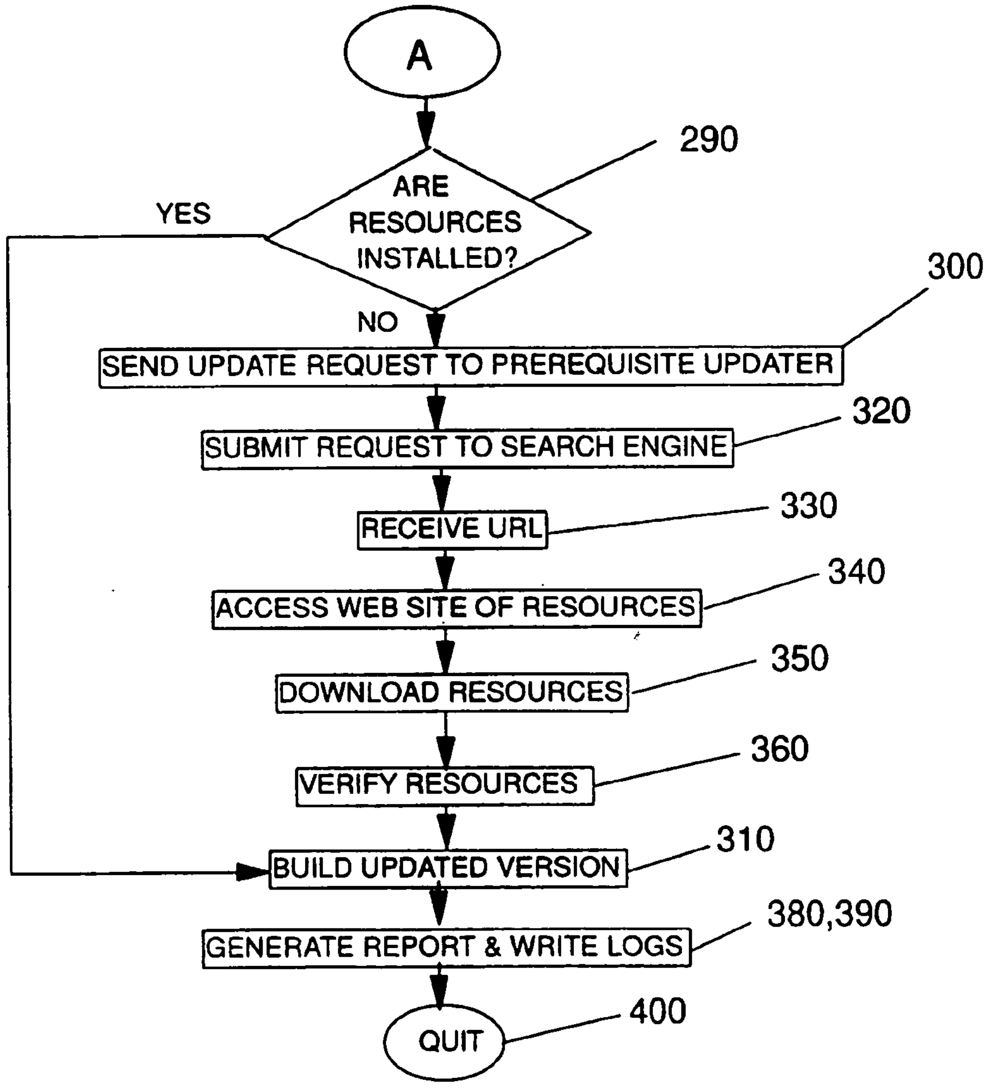

A
ARE RESOURCES INSTALLED?
290
YES
NO
SEND UPDATE REQUEST TO PREREQUISITE UPDATER
300
SUBMIT REQUEST TO SEARCH ENGINE
320
RECEIVE URL
330
ACCESS WEB SITE OF RESOURCES
340
DOWNLOAD RESOURCES
350
VERIFY RESOURCES
360
BUILD UPDATED VERSION
310
GENERATE REPORT & WRITE LOGS
380,390
QUIT
400

FIG. 4B

DISTRIBUTION OF SOFTWARE UPDATES VIA A COMPUTER NETWORK

FIELD OF INVENTION

The present invention relates to distribution of software via a computer network and to a mechanism for accessing software enhancements, corrections or new versions via a computer network. A 'network' of computers can be any number of computers that are able to exchange information with one another, and may be arranged in any configuration and using any manner of connection.

BACKGROUND

Software has conventionally been distributed in the form of programs recorded on a recording medium such as a diskette or compact disk. Customers buy the recording medium and a licence to use the software recorded on the medium, and then install the software onto their computers from the recording medium. The manufacture and distribution of the pre-recorded recording media are expensive, and this cost will be passed on to the customer. Also, the effort for customers of ordering or shopping for the software is undesirable.

The distribution cost is particularly problematic because most software products are frequently updated, both to correct bugs and to add new features, after the software has been delivered to the user. Some types of software products are updated many times each year. The cost of sending a new diskette or CD to all registered customers every time the software is upgraded or corrected is prohibitive and, although many customers want their software to be the most up-to-date, highest performance version and to be error free, not all customers want to receive every update. For example, the vendor may charge more for updates than the customer wants to spend, or new versions may require upgrading of other pre-requisite software products which the customer does not want to buy, or migrating to new versions may require migration of data which would disable the customer's system for a period of time.

Thus, software vendors tend to publicise the availability of new versions of their software and leave it for the customer to decide whether to purchase the latest upgraded version. For some software products, however, it is appropriate for the software vendor to proactively send out upgraded versions, or at least error correction and enhancement code (known as "patches") for their software products. Whatever a particular company's policy, significant costs and effort are involved in releasing these various types of software updates.

Increasingly, software distributors are using the Internet as a mechanism for publicising the availability of updates to their software, and even for distributing some software. The Internet is a network of computer networks having no single owner or controller and including large and small, public and private networks, and in which any connected computer running Internet Protocol software is, subject to security controls, capable of exchanging information with any other computer which is also connected to the Internet. This composite collection of networks which have agreed to connect to one another relies on no single transmission medium (for example, bidirectional communication can occur via satellite links, fiber-optic trunk lines, telephone lines, cable television wires and local radio links).

The World Wide Web Internet service (hereafter 'the Web') is a wide area information retrieval facility which provides access to an enormous quantity of network-

accessible information and which can provide low cost communications between Internet-connected computers. It is known for software vendors, customers who have Internet access to access the vendors' Web sites to manually check lists of the latest available versions of products and then to order the products on-line. This reduces the amount of paperwork involved in ordering software (and is equally applicable to other products). Some companies have also enabled their software to be downloaded directly from a Web site on a server computer to the customer's own computer (although this download capability is often restricted to bug fixing patches, low cost programs, and demonstration or evaluation copies of programs, for security reasons and because applying patches tends not to require any change to pre-requisite software or any data migration).

Information about the World Wide Web can be found in "Spinning the Web" by Andrew Ford (International Thomson Publishing, London 1995) and "The World Wide Web Unleashed" by John December and Neil Randall (SAMS Publishing, Indianapolis 1994). Use of the WWW is growing at an explosive rate because of its combination of flexibility, portability and ease-of-use, coupled with interactive multimedia presentation capabilities. The WWW allows any computer connected to the Internet and having the appropriate software and hardware configuration to retrieve any document that has been made available anywhere on the Internet.

This increasing usage of the Internet for ordering and distribution of software has saved costs for software vendors, but for many software products the vendor cannot just rely on all customers to access his Web pages at appropriate times and so additional update mechanisms are desirable.

As well as the problem of manufacture and distribution cost associated with distributing media, there is the problem that customers typically need to make considerable proactive effort to find out whether they have the best and the latest version and release of a software product and to obtain and apply updates. Although this effort is reduced when Internet connections are available, even a requirement for proactive checking of Web sites is undesirable to many users since it involves setting up reminders to carry out checks, finding and accessing a software provider's Web site, navigating to the Web page on which latest software versions and patches are listed, and comparing version and release numbers within this list with the installed software to determine whether a relevant product update is available and to decide whether it should be ordered. There may be an annoying delay between ordering an update and it being available for use, and even if the update can be downloaded immediately the task of migrating to an upgraded version of a software product can be difficult. If these steps have to be repeated for every application, control panel, extension, utility, and system software program installed on the system then updating becomes very tedious and time consuming. Therefore, manual updating tends not to be performed thoroughly or regularly.

There is the related problem that software vendors do not know what version of their software is being used by each customer. Even if the latest version of their software has been diligently distributed to every registered customer (by sending out CDs or by server-controlled on-line distribution), there is still no guarantee that the customer has taken the trouble to correctly install the update. This takes away some of the freedom of software developers since they generally have to maintain backward compatibility with previous versions of their software or to make other concessions for users who do not upgrade.

It is known in a client-server computing environment for a system-administrator at the server end to impose new versions of software products on end users at client systems at the administrator's discretion. However, this has only been possible where the administrator has access control for updating the client's system. This takes no account of users who do not want upgrades to be imposed on them.

Yet a further related problem is that software products often require other software products to enable them to work. For example, application programs are typically written for a specific operating system. Since specific versions of one product often require specific versions of other products, upgrading a first product without upgrading others can result in the first product not working.

"Insider Updates 2.0" is a commercially available software updater utility from Insider Software Corporation which, when triggered by the user, creates an inventory of installed software on a user's Apple Macintosh computer and compares this with a database of available software update patches (but not upgraded product versions) and downloads relevant updates. "Insider Updates" shifts the responsibility for finding relevant updates from the user to the database maintainer, but the access to update patches is limited to a connection to an individual database and the tasks of scanning the Internet and on-line services to find updates and of maintaining the database of available updates require significant proactive effort. "Insider Updates" does not install the updates or modify the user's software in any way. "Insider Updates" does not address the problem of unsynchronised prerequisite software products.

A similar product which scans selected volumes of a computer system to determine the installed software and which connects to a database of software titles for the Apple Macintosh, but does not download updates, is Symmetry Software Corporation's "Version Master 1.5".

An alternative update approach is provided by "Shaman Update Server 1.1.6" from Shaman Corporation, which consists of: a CD-ROM (updated and distributed monthly) that users install on a PowerMac file server; client software for each Macintosh computer to be inventoried and updated; and means for accessing an FTP site storing a library of current updates. "Shaman Update Server" creates an inventory of networked computers and downloads and distributes latest versions of software to each computer. Network administrators centrally control this inventory and updating process. The distribution of CD-ROMs has the expense problems described earlier.

SUMMARY OF INVENTION

According to a first aspect of the invention, there is provided an updater component for use in updating one or more computer programs installed on a computer system connected within a computer network, the updater component including:

information for identifying one or more locations within the network where one or more required software resources are located;

means for initiating access to the one or more locations to retrieve the one or more required software resources; and

means for applying a software update to one of said installed computer programs using the one or more retrieved software resources.

An updater component according to the invention preferably controls upgrading of, and fixing of bugs within, an associated software product or products automatically without requiring any interaction by the user after an initial agreement of update criteria. The update criteria can be associated with the products' licensing terms and conditions. This ensures that users who adopt a suitable update policy can always have the most up-to-date software available, with errors being corrected automatically from the viewpoint of the user. The user does not need to know where software updates come from, how to obtain them or how to install them since the update component takes care of this. The software vendor avoids having to ship special CDs or diskettes to correct errors or provide additional features; the vendor can easily release code on an incremental basis such that customers receive new product features sooner and with no effort.

An updater component according to a preferred embodiment of the invention performs a comparison between available software updates and installed software on the local computer system to identify which are relevant to the installed software, compares the available relevant updates with update criteria held on the local computer system (these update criteria are predefined for the current system or system user), and then automatically downloads and applies software updates which satisfy the predefined criteria.

This automatic applying of software updates preferably involves installing available software patches and/or upgraded versions in accordance with both the predefined update criteria and instructions for installation which are downloaded together with the program code required for the update. This feature of executing dynamically downloaded instructions provides flexibility in relation to the types of updates that can be handled by an updater component. It can also be used to enable a single generic updater component to be used with many different software products. Alternatively, the installation instructions for certain software updates may be pre-coded within the updater component. The "software resources" are typically a combination of program code, machine readable installation instructions and any required data changes such as address information.

The information for use in identifying a network location may be explicit network location information or it may be a software vendor name or any other information which can be used as a search parameter for identifying the location. In the preferred embodiment, the information is a product identifier which is provided by the updater component to a search engine to initiate a search to identify the relevant network location at which are stored the software resources for implementing updates to that product. This search may be performed by a conventional Internet (or other network) search engine which is called by the updater component. When the search engine returns an identification of the network location, the updater component retrieves from this location a list of available relevant updates, checks the list against the locally held software product version and against predefined update criteria, and retrieves the update resources onto the local computer system if those criteria are satisfied.

According to a preferred embodiment of the invention, a standardised naming convention is used for software resources from which to build software updates, and the updater component can search for these resources on a Network Operating System filesystem. This allows software resources to be stored at multiple locations to mitigate against network availability problems and makes it easier for developers and distributors to provide their error-fixing patches and upgraded versions of software products. For example, a developer can make new software updates available via a public network disk drive on their LAN using a known filename or via a published Uniform Resource Locator (URL) which can be searched for using known key words.

Updater components are preferably an integral part of the products they will serve to update. Hence, the updater component is distributed to software users together with an initial version of a software product, the updater component then automatically obtaining and applying software updates in accordance with preset criteria (such as a time period between successive searches for updates, and whether the particular user has selected to receive all updates or only certain updates—such as to receive updating patches but not replacement product versions for example).

The updater component's update capability preferably includes updating itself. Indeed, the update criteria may be set such that the updater component always accesses appropriate network locations to obtain updates to itself before it searches for software resources for updating its associated software products.

An updater component according to the invention preferably includes means for checking whether pre-requisite products are available, and are synchronised to the required version, as part of the process of selecting an update path for the current product. In a preferred embodiment, as well as checking their availability, the updater component is capable of instructing the updater components associated with pre-requisite software products to initiate updates to their software where this is the agreed update policy. If each software product's updater component is capable of triggering updates to pre-requisite products, then updates can ripple through the set of installed software products without the user having to be involved in or aware of the updates. This capability is a significant advantage over prior art updater agents which do not deal with the problem of unsynchronised software versions when one updates, and supports the increasing trend within the software industry for collaboration between distributed objects to perform tasks for the end user.

The updater component preferably also includes a mechanism for verifying the authenticity of downloaded software, using cryptographic algorithms. This avoids the need for dedicated, password-protected or otherwise protected software resource repository sites. The software resources can be anywhere on the network as long as they are correctly named and or posted to the network search engines.

Thus, the present invention provides an agent and a method for obtaining and applying software updates which significantly reduces the cost and effort for software distributors of distributing and tracking software updates and significantly reduces the effort for system administrators and end users of applying updates to installed software.

BRIEF DESCRIPTION OF DRAWINGS

The present invention will now be described in more detail, by way of example only, with reference to the accompanying drawings in which:

FIG. 1 is a schematic representation of a computer network including a local computer system having an installed updater component, server computers storing lists of available updates and storing software resources for applying updates, and a search engine for locating the servers;

FIG. 2 is an example of a software vendor's list of their software versions and the resources and prerequisites for building from one version to another;

FIG. 3 represents the sequence of operations of an updater component according to an embodiment of the invention; and

FIG. 4 is a further representation of the sequence of operations of an updater component.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

As shown in FIG. 1, an updater component 20 is installed in system memory of a conventional network-connected computer system 10, together with an associated computer program 30. The updater component may have been delivered to the user of the local computer system on a storage medium (diskette or CD) for him to install, or it may have been explicitly downloaded from another computer system. In preferred embodiments of the invention, updater components are integrated within the computer program they are intended to maintain (or are otherwise delivered via the same mechanism and at the same time as their associated program). The updater component is then installed as part of the installation procedure of its associated program, such that the user is not required to take any special action to obtain or activate it. The installation of each updater component includes the updater component registering itself with the operating system (more generally, updaters register with a repository 40, which may be central or distributed), such that at least the updater components on the local system are identifiable and contactable by address information, and/or their product identifier, within the register entry.

It is a feature of the preferred embodiment of the invention that each updater component can locate, can be located by, and can communicate with other updater components which manage other software products. This capability is used when one updater component requires another one to update to a specific level before the former can execute its own update, as will be discussed below. This is enabled by the updater components registering within the operating system or other repository 40.

In the preferred embodiment, each registration entry contains two items: the updater path and the updater network address. The path is the location of the updater component binary file so that the updater component can be launched by the operating system during the boot up process. This ensures that the updater component is always active and ready to perform work or handle requests issued to it from other updater components. The network address is the address used by components on other computer systems in the network to locate it on the network and to communicate with it.

An example of such registration using a UNIX (TM) operating system and the TCP/IP protocol suite uses the following naming convention for updater components: SoftwareVendorName+_product_name+_updater.

Path registrations can be entered in the UNIX/etc/inittab file to store the path entry. When, for example, the updater component for IBM Corporation's DB2 (TM) database product is installed it will add an entry to the /etc/inittab file of the form:

ibm_db2pe_updater:2:respawn:/usr/abin/db2_updater_binary

Every time the computer system reboots it will read this file and launch the DB2 updater component. The "respawn" keyword in the updater entry ensures that, should the updater component process fail for some reason during general system operation, it will be restarted by the operating system automatically. This approach will ensure that all updater components for all installed applications are always active.

Network Location registrations can be entered in the UNIX/etc/services file. For example when the DB2 updater component is installed it will add an entry to the /etc/services file of the form:

ibm_db2pe_updater 5000/tcp #net location of DB2 updater component

When another updater component wishes to communicate with the DB2 updater component it will find it by searching this file for the DB2 updater component name ibm_db2pe_updater (actually done indirectly by the UNIX call getservbyname()—the name is built by the caller according to the standard naming convention). When it is found it knows that the DB2 updater is listening for connections on port number 5000 and will use the TCP protocol. This allows the updater component in question to establish a link to the DB2 updater component and start a conversation (described later).

For an updater component to find and talk to another updater component on another remote machine the above information would have to be augmented by having a repository **40'** which is accessible from both machines (preferably a central or distributed database accessible from anywhere in the network, such as a Web Page or pan-network file) and is available to all updater components that require it. Entries would be of the form updater_name machine_ip_address (OR DNS entry), port number, protocol.

For example, the manufacturing department of an organisation may have three computer systems on which distributed software products collaborate with each other, the systems being called a, b and c. Typical entries in the Web page or file manufacturing_collaborators.html might be:

ibm_catia_updater a.manufacturing.com 5000 tcp
ibm_db2pe_updater b.manufacturing.com 5100 tcp
ibm_cics_updater c.manufacturing.com 4780 tcp

An updater component can then connect and talk to any other updater component using the DNS name to create an IP address and the port number which the remote updater component is listening to at that address.

The steps of updater registration at installation are therefore:

1) Create entry in /etc/inittab file (register updater process code location)
2) Create entry in /etc/services file (register updater process local address)
3) Create entry in central database file (register updater process pan-network address).

The installation process may also involve providing to the updater component the local IP address of a Web proxy server. It will be clear to persons skilled in the art that many alternative registration implementations are possible.

Updater components include data fields for an identifier and version number for their associated software products. The updater components may be delivered to customers with these fields set to null values, and then the installation procedure includes an initial step of the updater component interrogating its software product to obtain an identifier and current program version and release number. Alternatively, the software vendor may pre-code the relevant product ID and version number into the updater component.

The system **10** of FIG. **1** is shown connected within a network **100** of computers including a number of remote server systems (**50,50'**) from which software resources are available for applying updates to programs installed on the local system **10**. Each server system includes within storage a list **60** of the latest versions of, and patches for, software products which are available from that server. Each vendor is assumed here to make available via their Web sites such a list **60** of software updates (an example of which is shown in FIG. **2**) comprising their product release history, in a format which is readable by updater components, and to make available the software resources **70** required to build the releases from a given level to a new level (this transition

from a software product release to a new level will be referred to hereafter as a 'growth' path). The entries in the software updates list **60** include for each software product version **110** an identification **120** of the software resources required for applying the update and an identification **130** of its prerequisite software products and their version numbers. In some cases, the required resources are complete replacement versions of software and associated installation instructions. In other cases, the resources comprise patch code for modifying an existing program (e.g. for error correction) and the patch's installation instructions.

For the current example, we will assume that the network **100** is the Internet, although the invention may be implemented within any computer network. Also shown within the network **100** is a server system **80** on which a search engine **90** is installed for use in finding update source locations on the network. This is shown located remotely from the local system **10**, although it need not be. In the Figure, each updater component **20** is shown associated with a single program **30**, and it is a feature of this embodiment of the invention that all installed software products have associated updater components which manage them, but neither of these features is essential to the invention as will be explained later.

The operation of an updater component will now be described, with reference to FIGS. **3** and **4**. When an installed updater component executes, its first action is to initiate **200** a search for available updates to the particular software product, providing to one or more search engines **90** as search arguments the product identifier and product version release number obtained at install time. Assuming that software vendors provide via their Web sites a list **60** of available product updates referenced by product identifier and release number **110** (or some other consistent naming convention is used), the search should identify the relevant Web site **140** on which update information is available. If the initial attempt to start a search engine is unsuccessful, then the updater component will attempt to start a different search engine (which may be in a different geographical location to the first), but could alternatively wait for a preset time period and then retry. A URL identifying the relevant Web site **140** for update information is returned **210** to the updater component as a result of the search.

The updater component uses the URL to access **220** the list **60** and downloads **230** a file **160** comprising the portion of the list **60** of available updates which relates to the particular product. The updater component then performs steps **240**-**280** as shown in FIG. **4**. Each file **160** contains message digests (e.g. MD5) which are digitally signed. The retrieved file **160** is then analyzed **240** using a digital signature checking algorithm (such as the algorithm described in U.S. Pat. No. 5,231,668). This is important to verify that the file **160** represents the correct software updates list for the particular software product, and that the file has not been tampered with since signing. Also, checking for the digital signature is a useful way of filtering the results of the search since these may include a plurality of Web page URLs other than the correct one (the search may find other pages which have a reference to the named product version, including pages not published by the software vendor). If an attempt to download and verify a file is not successful, then the updater component moves on to the next URL found in the search.

The updater component then performs on the local computer system a comparison **250** between the current installed software product's identifier and release number and the listed available updates in the retrieved file **160**. This comparison determines possible growth paths from the current to updated versions, but these possible growth paths are then compared 260 with predefined update criteria, and any possible paths which do not satisfy the update criteria are discarded. Thus, the updater component determines whether it is possible to migrate from a current software product to the available new versions and whether it is possible to apply patches to the current version under the currently agreed licence terms and conditions.

For example, the software product licence may enable migration to any future version of the product and application of any available patches, or only migration up to a specified version, or it may only permit applying of available patches which modify or correct errors in the current version. Possible update paths which are unavailable due to current license limitations are notified 270 as a system generated message sent to the software asset manager (who may be an end user or IT procurement manager) of the currently installed version, to enable them to make decisions about whether the current licence is adequate.

As well as licence restrictions as to the updates that are possible, an updater component's update criteria or growth policy includes a cycle period (for example weekly or monthly) and criteria for determining which of a plurality of possible growth paths to select (such as always select latest version permitted by licence, or always select latest patch and only notify availability of new versions, or only select new versions if prerequisite software is already available on local system). The growth criteria may also include control information such as when to upgrade to new versions that are downloaded by the updater component—if data migration is required when migrating to a new software product version it may be essential for this to be done outside of office hours or only at a single scheduled time each month or each year and this can be controlled by the updater component.

The growth policy definition may also include a parameter determining that updating of pre-requisite software products should be requested when required to maintain synchronisation with the current product. This will be described in more detail below. Persons skilled in the art will appreciate that there is great flexibility in the criteria that can be set and applied by the updater component.

The updater component then decides 280 on a particular growth path (i.e. which available version to upgrade to) from the set of possible growth paths using the update criteria. For example, the updater component may select the highest possible version or release number of the available updates which is permitted by the update criteria, if that is the update policy.

The updater component performs 290 (see FIGS. 3 and 4) a scan of the operating system file system to check whether the required software resources are already available on the local computer system. The required resources are the software update artifacts required to bring the current application software to the new level, and the software updates required for updating pre-requisite software to required levels. Each updater component associated with pre-requisite installed products is contacted 300 to ensure (a) that it is installed, and (b) that it is at or greater than the required pre-requisite level. If all required resources are available locally or on another machine (in the case of software relying on some pre-requisite software operating on a remote machine), and have been verified, then the updater component progresses to the step 310 (see FIG. 4) of building the updated software version. If not, the update component must obtain the required resources.

As shown in FIGS. 3 and 4, if required software resources for building the updated version are not found on the local system, the updater component submits 320 a further request to one or more search engines to find the required resources. The search engine returns 330 one or more URLs and the updater component uses these to retrieve 340,350 the software resources into storage of the local computer system. At this stage, the updater component or the user need not have any knowledge of what corrections or enhancements may be included in the new version—the update criteria determine what type of updates are required such that the user is spared the effort of studying the content of every update. In practice, it is desirable for users to be able to determine the effects of updates and so the software resources for the update include a description of these effects which a user or administrator can read.

As examples, the software product to be updated may be a word processor application program. If the word processor as sold missed certain fonts or did not include a thesaurus, patches may subsequently be made available for adding these features. The updater component has the capability to add these to the word processor, subject to the update criteria.

In alternative embodiments of the invention, the search for required software resources is unnecessary following the initial search for the updates list (or is only necessary where there are pre-requisite software products as well as patches or new versions for the current product—see below). This is because the update software resources required directly by the current product are stored in association with the list of required resources. That is, the list includes a pointer to the network location of the required resources such that a selection of a growth path from the list involves a selection of a pointer to the network location of the required updates (and possibly also pointers to the locations of pre-requisite software products).

A second verification by digital signature checking is performed 360 (see FIG. 4), this time on the downloaded resources. After verifying 360 the legitimacy of the downloaded resources, the updater component automatically builds 310 the installation in the target environment in accordance with the update policy. In practice, this may require information from the user such as an administration password, or a database usage parameter value, but in the preferred embodiment of the invention installing of the downloaded code is automatic in the sense that it does not require the user to know or obtain from elsewhere any installation information and in that it generally enables the user to be freed from making any decisions at run time if the predefined update criteria enable the updater component to automatically apply updates.

It is well known to include machine readable installation instructions encoded in a shell (for example as Script, or an interpretive language such as PERL, or an executable such as setup.exe in the case of applications on Microsoft's Windows (TM) operating system). Updater components according to the invention will download 350 the machine readable instructions together with the relevant software resources and will automatically execute them 310. The updater component thus automatically processes installation instructions, avoiding the input from a person which is conventionally required. The Scripts can be adapted to reuse information gleaned from the first human installer who installed the first version of the updater component (for example, information such as user name and password of application administrator, installation directory, etc).

The method of updating according to the preferred embodiment of the invention requires software vendors to organise the software resources required to build from one product level to another. For example, a move from version 1.1.1 to 1.1.4 would typically include a series of patches to be applied, and the required order of installation if any would advantageously be encoded in machine processable installation instructions. The user is then spared the effort and the risk of human error which are inherent in methods which require the user to control the order of application of fixes and enhancements. The problem of how to migrate from one product level to another is thus dealt with by the software vendor instead of the customer, and updater components can only move to levels supported by the vendor (i.e. those growth paths published by the software vendor for a specific existing product level).

The updater generates **380** a report and writes **390** to log records, and then quits execution **400** (in the preferred embodiment the updater goes into a sleep or idle state) until activated again **410** upon expiry of a predetermined update cycle period (the repeat period parameter is configured when the updater component is installed).

Structure of Updater Component

The structure of an updater component comprises data, methods for operating on that data, and a public application programming interface (API) which allows other updater components to contact and communicate with it. This structure will now be described in detail.

UPDATER COMPONENT DATA:

The updater component includes the following persistent data:

Product_ID: an identifier of the software product which is managed by this updater component

Current_Installed_Version: a version identifier for the installed software (e.g. version 3.1.0)

Current_License: a version identifier corresponding to the software product version up to which the current software license allows the user to upgrade (e.g. version 4.0.z). Alternatively, this may be a licence identifier (e.g. LIC1) for use when accessing machine readable licence terms.

Installation_Environment:

a list of attribute name/attribute value pairs.

This is used by the updater component to store values entered by the user when the updater was used for the first time. For example, the updater installation userid and password, possibly the root password, the installation directory, the web-proxy server address, search engine URLS, log file name, software asset manager e-mail address etc. This data will be re-used when subsequent automatic updates are required.

Growth policy parameters:

a. Growth_Cycle: data determining whether the updater component should attempt to update its software product every day, week or month, etc.

b. Growth_Type: data determining whether the updating is limited to bug fixing and enhancements (i.e. patches) only or requires upgrading to the latest release in each growth cycle.

c. Force_Growth: (YES/NO) a parameter determining whether to force other software resources to upgrade if that is a pre-requisite for this software to upgrade. (Some implementations will provide more flexible controls over forcing other software to update than this simple YES/No)

Last_Growth_Time: Date and time when updater component last executed

The updater component also includes the following non persistent data:

Possible_Growth_Paths:

transient data representing the available upgrade paths (e.g. version numbers 3.1.d, 3.2.e, 4.0.a)

PRIVATE UPDATER FUNCTIONS:

The updater component logic includes the following methods:

Discover_Possible_Growth_Paths()

Search for Growth_Path information for this software product on the Internet (or Intranet or other network). This search method initiates a search via a standard search engine server. The information returned is a list of newer versions and associated pre-requisite product information.

The Growth_Path information is then reduced in accordance with the Growth policy parameters. For all members in the Growth_Paths list, a check is performed of whether appropriate versions of pre-requisite products are available on the local and/or remote computer. The updater components managing these pre-requisite products are accessed and forced to grow if this is the policy.

If all pre-requisite products exist locally at the correct level, or are available remotely on the network and there is with a "force growth" policy, then identifiers for newer versions of the software product are added to the Possible_Growth_Paths list.

Decide_Growth_Path()

Interpret the growth policy and select a single growth path. Some implementations of the invention will involve user interaction to select the path, for example if there are considerations such as whether to force updates to other programs.

Get_Resources(Parameter: Chosen_Growth_Path)

Given Chosen_Growth_Path (e.g.3.2.0), search for required resources (Parameters Product_ID, Current_Installed_Version, Chosen_Growth_Path), download all resources to local computer. This will include software required for the new version plus machine processable installation instructions.

Install_Resources()

Process installation instructions including installing required files in correct locations, possibly compilation of the files and modifying the configuration of the existing system to accommodate the software, logging all actions to a file (and enabling an "uninstall" method to undo all actions).

Grow()

Initiates methods:

```
Discover_Possible_Growth_Paths()
if no possible growth paths exist then updater component
    becomes idle else
Decide_Growth_Path()
Get_Resources(Parameter: Chosen_Growth_Path)
Install_Resources().
```

Then Growo writes all completed actions to log and finishes execution of the updater component. The updater component becomes idle either until time to check again for new update requirements or until prompted by another updater component to do so.

PUBLIC UPDATER COMPONENT API:

The updater component includes the following public API. These functions would be callable using existing network communications software, such as remote procedure calls, message oriented middleware, ORB (Object request broker), etc.

Get_Release()

This function is called by other updater components and returns the release level of the product managed by this updater component.

Update(new_level)

Other updater components call this function to move the product managed by this updater component to a new level indicated by the new_level parameter value. This will call the private function Grow().

Receive_Event(event details)

When an updater component receives a request to update, it must inform the calling updater component when it has completed the update or otherwise e.g. if it failed for some reason. The updater component performing the update on behalf of another updater component will call this function of the requesting updater component to communicate success of the update or otherwise. Event details can be a string like "product id, new release level, ok" or "product id, new release level, failure".

The automatic handling of the potential problem of unsynchronised pre-requisite products by enabling forcing of updates (or, if forcing of updates is not part of the update policy, sending of notifications to the software asset manager) is a significant advance over prior art update schemes.

Since the updates list file **160** returned to the updater component in response to an initial search includes an identification **130** of pre-requisite software, that information enables the aforementioned examination **290** of the updater component registration database **40,40'** to check whether pre-requisite software is available locally or remotely. If it finds all the updater components located locally or remotely, it can be sure that the software pre-requisites are available and it next needs to contact each updater component for each software product to be sure all pre-requisites are at the correct level. If an updater component **20'** having a required product identifier for pre-requisite software **30'** but not having the required version number is found locally or remotely, and if forcing of updates is the update policy, then the updater component **20** of the first computer program contacts **300** this pre-requisite updater component **20'** and requests that it attempt to update its associated pre-requisite software product **30'**. This updater component **20'** can, if necessary, request other updater components of its pre-requisite software to update their versions, and so on.

If at some stage no relevant updater component is found locally or remotely, then a message is sent to the asset manager to inform him/her of the requirement for a new product in order to grow the associated product further. If at some stage during the chain of updater requests to grow to a new level one updater component fails to move to the required level then this failure is reported back to its calling updater component, prompting failure of that components update operation, and so on back to the updater component which initiated the whole transaction.

Thus, as well as their autonomous behaviour defined by their update criteria, updater components can react to external stimuli such as requests from other updater components.

Example of Update Synchronisation

An example of the implementation of update synchronisation between two products will now be described. This example shows how one updater component can communicate with another to synchronise pre-requisite software so that all products are present and at compatible release levels.

A CORBA (Common Object Request Broker Architecture) ORB (Object Request Broker) is used for location of and communication between two updater components. Using the above public API it is a simple matter for those familiar with the art of CORBA programming to develop communication code so that one updater component

can talk to another updater component anywhere on a network. In this example the component updater registration database **40** is a directory or folder available over the network (e.g. via NFS) which contains for each installed updater component a file called "updater-component_name.iop" (iop stands for interoperable object reference).

This file contains a sequence of bytes which can be converted into a reference to the updater component by any updater component which reads the file using for example the CORBA function:

CORBA::Object::_string_to_object() in C++

Furthermore this reference can be to an updater component anywhere on the network as it represents a unique address for the corresponding updater component. When updater component A has manufactured a reference to updater component B then updater component A can call a public API function simply by using, for example, a C++ mapping A→Get_Release() which will then return the value of the release level of the software managed by the A updater component.

In this example we will consider two products—IBM Corporation's DB2 database product and a Query Tool called "Query Builder", on different machines M and N respectively. (Machines M and N could be the same machine; the present example merely shows that they may also be separate). Both products have updater components which use a CORBA ORB architecture as briefly outlined above. An ORB communication daemon is active on participating systems M and N.

Step 1) Registration Phase:

The DB2 Updater Component starts when the operating system starts on system M and immediately creates a file called ibm_db2_updater.iop (according to some naming standard used to aid subsequent searches for the file) in the network file system folder or directory. This directory could be hosted on any machine and not necessarily M or N. The file contains a series of bytes which can be used to manufacture a reference to the updater component.

```
[pseudocode]
Filehandle=open("/network/filesystem/directory", "ibm_
    db2_updater.iop");
ReferenceBytes=CORBA::Object::_object_to_string();
Write(FileHandle, ReferenceBytes);
close(Filehandle);
```

QueryBuilder Updater component starts and writes its registration to the same directory or folder, again in this case calling the file ibm_querybuilder_updater.iop.

At this stage both updater components are active and have registered their presence and location in the network directory.

Step 2).

QueryBuilder attempts to grow from version 1 to version 2 but a prerequisite is DB2 version 2.1 or higher. The following sequence of actions will occur. QueryBuilder is denoted QB and DB2 as DB2.

QB: searches for file ibm_db2_updater.iop (file name manufactured according to standard) in network directory. It finds the file, reads it and converts it to a usable reference.

```
[pseudocode]
if (dbref=CORBA::Object::_string_to_object(readfile
    (ibm_db2_updater.iop)))
  then SUCCESS we have connected to the updater else
  FAIL: Prerequisite software does not exist in set of
    collaborating systems—send e-mail to software asset
    manager to notify situation.
```

```
Give up on trying to grow to new version.
endif.
```

Step 3).

At this stage we know that DB2 exists somewhere in our set of networked computers. Now we need to know if it is at the right level. We simply do this by executing its public API function Get_Release() defined above, from within the QB updater, the QB updater is therefore a client requesting the DB2 updater to do something for it, i.e. tell it what release it is.

```
[pseudocode]
db2_release=dbref→Get_Release();
```

Let us say this returns the value "2.0".

Step 4)

Client Side:

The QB Updater Component knows that this is not sufficient , it requires version 2.1. It examines its Force_Growth parameter which is, for example, "YES" meaning it should force pre-requisite software to grow to the level required before it can perform its own update procedure. Therefore the QB updater tells the DB2 updater to grow to the new release, and then waits until the pre-requisite has grown to the new release or failed in doing so.

```
[pseudocode]
dbref→Update("2.1", QBref); // QBref is a ready made
    reference to the // QB Updater. It is passed to the DB2
    updater so that // it can quickly send the results, success
    or failure, // when the DB2 Updater has finished trying to
    update // itself.
EVENT=null;
While (EVENT equals null)
    {do nothing;}
if (EVENT equals "SUCCESS")
    then attempt to grow software managed by this updater
        component i.e. Query Builder.
else
    Write failure to log;
    do not attempt to grow;
    go to sleep and try later;
endif.
```

Server Side:

The DB2 Updater component receives the request to grow. Which it attempts to do.

It reports the result to the calling client (it knows how to contact the calling client as it is receives a reference to the caller in the function call.)

```
[pseudocode]
DB2 attempts to grow.
if Growth Successful then
    QBRef→Receive_Event("SUCCESS"); // Note the
        implementation of the // function Receive_Event sim-
        ply sets the variable // called EVENT in the QB
        Updater component to the // value of the parameter
        passed in the API call , i.e. // "SUCCESS" if in this
        section of the IF statement.
else
    QBREF→Receive_Event("FAILURE");
end if
```

As noted previously, predefined update criteria may determine which of an available set of updates should be applied and which should be disregarded. The update criteria may include an instruction to the updater component to send a notification to the end user or system administrator when a software update is identified as being available but applying this update is not within the update policy or is impossible. One of the examples given previously is that the update

policy may be not to install full replacement versions of software products since that may require upgrading of pre-requisite software products or migration of data (for example if the software product is a database product), whereas it may be intended policy to install any error-correction patches. Notification rather than automatic installation of updates may also be implemented where to upgrade one product to a new version would require upgrading of other pre-requisite complementary products.

The update policy can also determine the degree of automation of the updating process, by defining the circumstances in which the updater requests input from the user or administrator.

The execution of a particular example updater component will now be described in more detail by way of example. This updater component's function is to keep an installed product called "Test" totally up-to-date with all released patches, but not to install replacement versions of Test. Firstly, the updater component is configured with the following data instantiations:

Product_ID: Test
Current_Installed_Version: 1.0.a
Current_License: LIC1
Installation_Environment:"USERID:TestOwner, USERPASSWORD:easy"
"INSTALLPATH: /usr/bin/testapp/"
Growth_Cycle: weekly
Growth_Type: patches, latest, automatically
Force_Growth: no
Last_Growth_Time: Monday Aug. 10, 1997.

The updater then executes weekly, for example each Monday night at 3 am (it is the system administrator who decides the timing).

The following represents a possible execution trace for this example updater component.

Example Execution Trace

Step 1) The Growth Cycle Starts:

>>>> START: Discover_possible_Growth_Paths()

* Execute search on remote search engine (e.g. Internet Search Engine) using Phrase ("IBM Test 1.0.a Growth Paths")

Search returns URL published by software vendor outlining current growth paths for product;

* Download URL:

File contents are:

"1.0.b,none; 2.0, other_required product_product_id 1.0.c;"

* Authenticate URL file using hashing algorithm and digital signature.

If not authentic, return to search for another URL matching criteria

* Build growth_path_list: growth_path list="1.0.b, none;

2.0, other_required_product_id 1.0.c;"

* Remove all but patch level increases (according to Growth_Policy) from Growth_path list (i.e. only those with the first version and second release number matching 1.0).

* growth_path list="1.0.b, none;"

* For all members in list, ensure prerequisites exist. In this example, all members of list meet this criteria trivially.

* Place candidate growth_paths into Possible_Growth_Paths list=1.0.b

<<<< END: Discover_possible_Growth_Paths()

Step 2) Next the updater component decides on the Growth Path to pursue:

>>>> START Decide_Growth_Path()

* The growth policy dictates that we should grow to latest patched

revision. (In this example, determining the latest revision is trivial i.e. it is 1.0.b)

* chosen_growth_path=1.0.b

<<<< END: Decide_Growth_Path()

Step 3) The updater component then obtains the required resources to revise the current software level to the new one.

>>>> Get_Resources()

* Execute search on remote search engine (e.g. Internet Search Engine) using Phrase ("IBM Test REVISION 1.0.a to 1.0.b

RESOURCES").

* Search returns URL say

ftp://ftp.vendor-site/pub/test/resources/1.0.a-b"

* Updater downloads file pointed to by URL and places in secure holding area where it verifies authenticity.

* Updater verifies authenticity (using, for example, digital signatures based on RSA algorithm, or any method)

If files not authentic, then return to search (see Note 1 below)

* Updater unpacks resources into a temporary directory (see Note 2

below). These resources include machine processible installation

instructions (for example, instructions written in a script

language such as a UNIX shell script or MVS REXX) and files

(either binary or requiring compilation) which actually contain

the software fix.

<<<< END: Get_Resources()

Notes on above tasks

Note 1—To save time the updater looks for a standard file before downloading the URL called "signature", which contains the URL

ftp://ftp.vendor-site/pub/test/resources/1.0.a-b and a listing of its contents. This is hashed and signed. Using this signature, the Updater component can quickly establish authenticity of the URL (to some extent) before downloading it and use the information i.e. file listings to corroborate the final downloaded resources after they have been unpacked into the temporary directory. When the final URL is downloaded it is also checked again for authenticity (to guard against someone placing a bogus artefact in an authentic URL location).

Note 2—Part of the unpacking is that the updater component will examine the installation scripts and modify them based on the contents of its installation environment data where required. For example if the installation instructions were coded in a shell script it will replace all instances of INSTALLPATH with the token "/usr/bin/testapp/". Again Naming conventions of attributes are standardised as it the method of token substitution in installation instructions. This makes totally automatic installation possible.

Step 4) The updater component then implements the actual software upgrade:

>>>> START Install_Resources()

* execute the installation instructions.

* update the values of

Current_Installed_Version=1.0.b

Last_Growth_Time =Date+Time.

* send an e-mail to software asset manager informing of

installation and whether or not a reboot of the Operating System

or restart of the application is required before the upgrade takes affect.

<<<< END Install_Resources()

This is the end of this current growth cycle. The seed updates the Last_Growth_Time value the current time and then exits. The time taken for this cycle could be anything from a few seconds where the updater component found no upgrade paths for the currently installed version to several hours if a totally new release from the current one is to be downloaded and installed together with new pre-requisite software.

An alternative to the embodiment described above in detail does not require an independent updater component for every different software product, but uses a single generic updater component installed on a system together with product-specific plug-in objects and instructions which are downloaded with each product. These objects interoperate with the generic code to provide the same functions of the product-specific updater components described above. It will be clear to persons skilled in the art that the present invention could be implemented within systems in which some but not all application programs and other software products installed on the system have associated updater components, and that other changes to the above-described embodiments are possible within the scope of the present invention.

What is claimed is:

1. A computer program product, comprising computer program code recorded on a computer readable recording medium, the computer program code comprising an updater component for use in updating one or more computer programs installed on a computer system connected within a computer network, the updater component including:

means for initiating access to one or more identifiable locations within the network where one or more required software update resources are located, to retrieve the required software update resources;

means for performing a comparison between software update resources available from said one or more identifiable network locations and computer programs installed on said computer system, to identify available relevant update resources, and for comparing the available relevant update resources with predefined update criteria corresponding to applicable software licence terms and conditions;

means for initiating retrieval of software update resources which satisfy said predefined criteria; and

means for applying a software update to one of the installed computer programs using the one or more retrieved software resources.

2. A computer program product according to claim 1, wherein said means for applying software updates includes means for installing available relevant software resources in accordance with the predefined update criteria and in accordance with computer readable instructions for installation which are part of the software resources downloaded for the update.

3. A computer program product according to claim 1, wherein information for identifying one or more locations is

held by said updater component and includes a product identifier of a computer program product, the updater component being adapted to provide said product identifier to a search engine, the product identifier serving as a search parameter for use by said search engine to identify network locations.

4. A computer program product according to claim 3, wherein said updater component is adapted to download a list of available software update resources and their pre-requisite software products in response to said search engine identifying network locations at which said list is held, to compare the list of available software update resources and pre-requisite products with computer programs installed on said computer system and, where updates to the pre-requisite products are required, to request updates to the pre-requisite products.

5. A computer program product according to claim 1, wherein the updater component has machine readable installation instructions for installing the updater component on a computer system, the installation instructions including instructions for registering the updater component with a repository which is accessible by other updater components, such that the updater component is identifiable and contactable by other updater components.

6. A computer program product according to claim 5, wherein the updater component includes an API via which updater components of complementary computer programs can request that the current updater component update its computer program, the current updater component being adapted to call an update method to update its computer program in response to an update request, and wherein the current updater component is adapted to send a system-generated request to updater components of pre-requisite computer programs of its computer program when updating of its computer program requires updating of said pre-requisite computer programs.

7. A computer program product according to claim 12, wherein said means for applying updates is adapted to install correction and enhancement software which modifies existing installed software and also to install upgraded versions of installed software which replaces installed software.

8. A method for automated updating of a computer program installed on a computer system connected within a computer network, including the following steps:

delivering to the computer system an updater component for use in updating the computer program;

providing at a first network location downloadable software resources for building said computer program from a current version to an updated version;

wherein the updater component is adapted to perform the following steps when executed on the computer system:

(a) initiating access to said first network location at which said software resources are located;

(b) performing a comparison between software resources available from said first network location and the installed computer program, to identify available relevant update resources, and comparing the available relevant update resources with predefined update criteria corresponding to applicable software licence terms and conditions;

(c) downloading onto said computer system the available relevant software update resources which satisfy the predefined update criteria;

(d) building said computer program from the current version to the updated version using the downloaded software resources.

9. A method according to claim 19, including providing at a second network location, identifiable from information in the updater component, a computer readable list of available updates to said computer program, wherein the updater component is adapted to perform the following steps prior to accessing said first network location:

initiate access to said second network location to retrieve said list;

read said list and perform a comparison of the listed available updates with said computer program on said first computer system, thereby to identify the available relevant update resources.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: 6,199,204 B1	Page 1 of 1
DATED	: March 6, 2001	
INVENTOR(S)	: Seamus Donohue	

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [75], change item [75] from "Inventor: **Seamus Donohue**, Artane (IR)" to
-- Inventor: **Seamus Donohue**, Dublin, Ireland --.

Signed and Sealed this

Twenty-fourth Day of September, 2002

Attest:

[illegible]

JAMES E. ROGAN

Attesting Officer

Director of the United States Patent and Trademark Office

US006202091B1

(12) **United States Patent**
Godse

(10) **Patent No.: US 6,202,091 B1**
(45) **Date of Patent: Mar. 13, 2001**

(54) **PROCESS AND APPARATUS FOR INITIALIZING A COMPUTER FROM POWER UP**

(75) Inventor: **Dhananjay Godse**, Kanata (CA)

(73) Assignee: **Nortel Networks Limited**, Montreal (CA)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/986,783**

(22) Filed: **Dec. 8, 1997**

(51) **Int. Cl.**^{7} **G06F 15/177**; G06F 13/38

(52) **U.S. Cl.** **709/222**; 709/220; 713/2

(58) **Field of Search** 709/200, 203, 709/217, 218, 219, 220, 221, 222; 713/2, 1, 100

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,452,454 *	9/1995	Basu	713/2
5,819,107 *	10/1998	Lichtman et al.	713/2
5,838,910 *	11/1998	Domenikos et al.	709/203
5,842,011 *	11/1998	Basu	713/2

OTHER PUBLICATIONS

"IPL a Remote LAN Workstation Using OS/2 and NETBIOS", IBMTDB vol. 33, 8 (NN910198), Jan. 1998.*
"Multiple Bootable Operating System", IBMTDB vol. 35, 1A (NA9206311) pp. 311–314, Jun. 1992.*
"Support of Multiple initial Microcode Load Images on Personal Computer DIskettes", IBMTDB vol. 36, 10 pp. 47–54, Oct., 1993.*

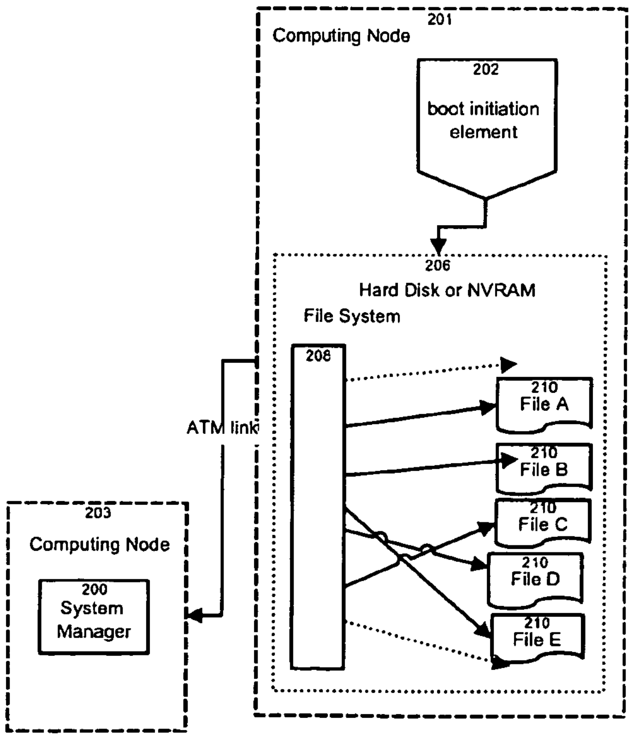
* cited by examiner

Primary Examiner—Mark Rinehart
Assistant Examiner—Paul Kang

(57) **ABSTRACT**

The present invention relates to a method and apparatus to permit a computer to boot from power up from its own memory units or from a network. The booting process is divided into the five stages of start-up, discovery, software download, software initialization and datafill. A boot ROM directs the stages in the booting process and the policy governing each of these five stages is laid out in a policy file. Each of the stages may be driven by the computer node's local memory unit, such as a local hard disk or non-volatile RAM), or from a software system downloader and configuration manager situated elsewhere in an ATM network where the computer node resides. A failure recovery procedure is also provided to allow the computer to revert to other means in case of failure during the booting process. The invention also provides a machine readable medium comprising a program element to implement the novel boot-up procedure.

27 Claims, 12 Drawing Sheets



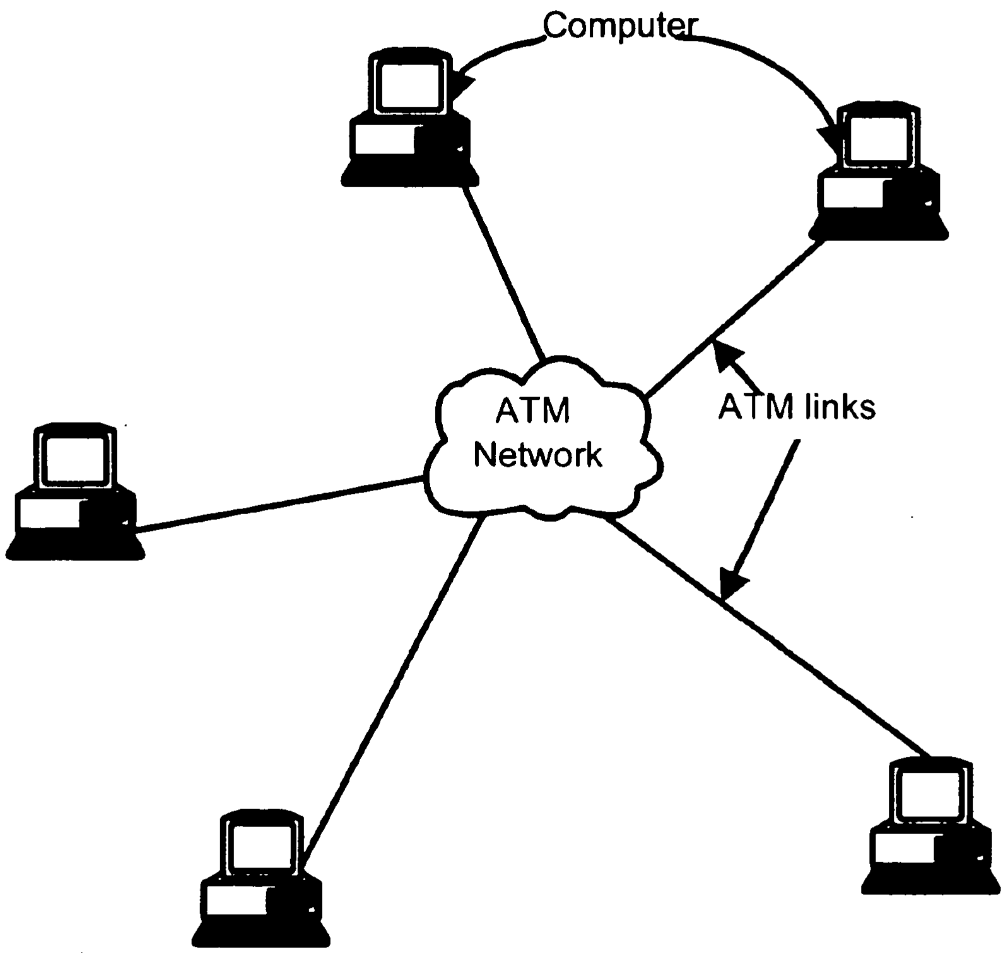


Fig. 1

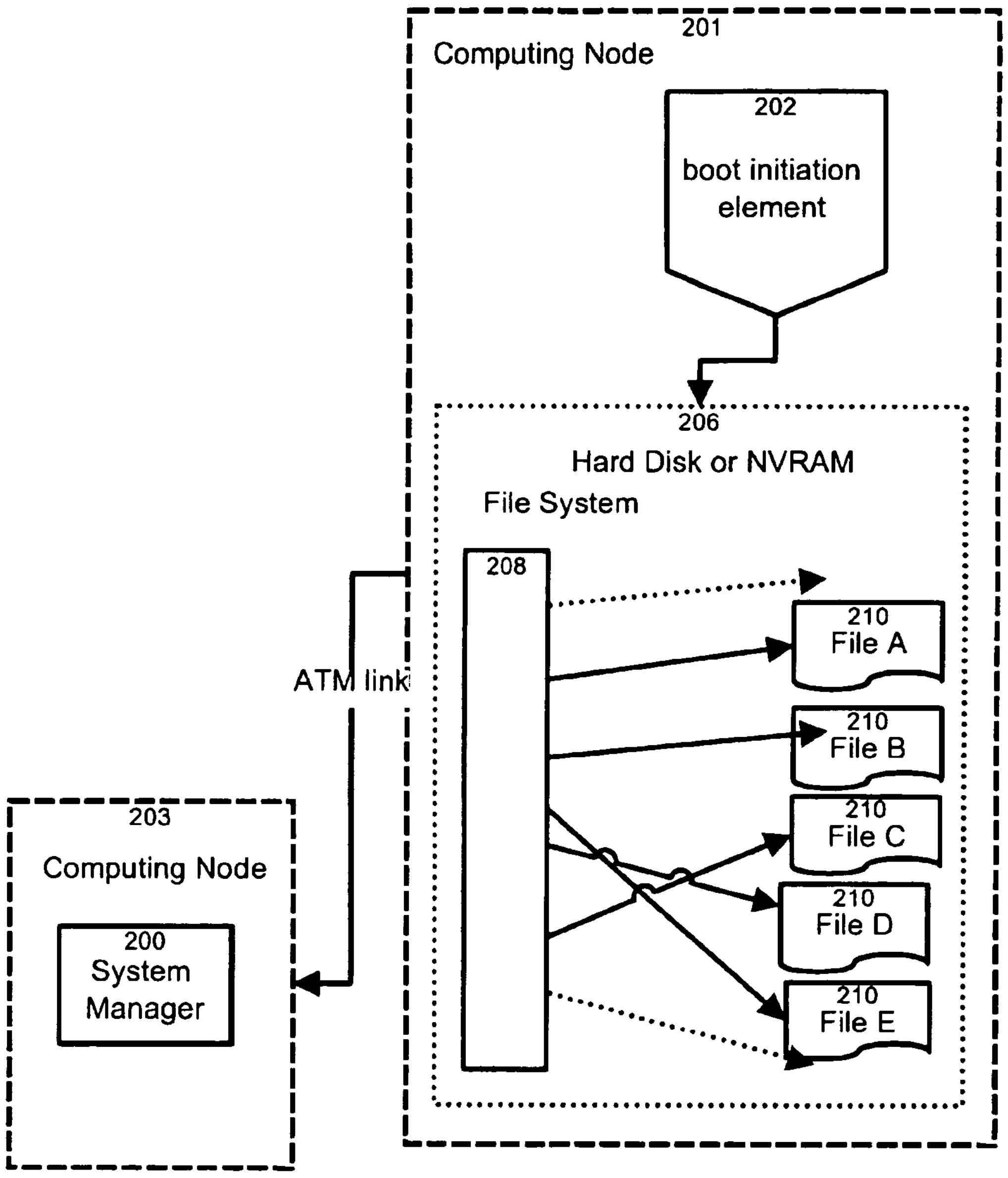

201
Computing Node
202
boot initiation element
206
Hard Disk or NVRAM
File System
208
210
File A
210
File B
210
File C
210
File D
210
File E
ATM link
203
Computing Node
200
System Manager

Fig. 2

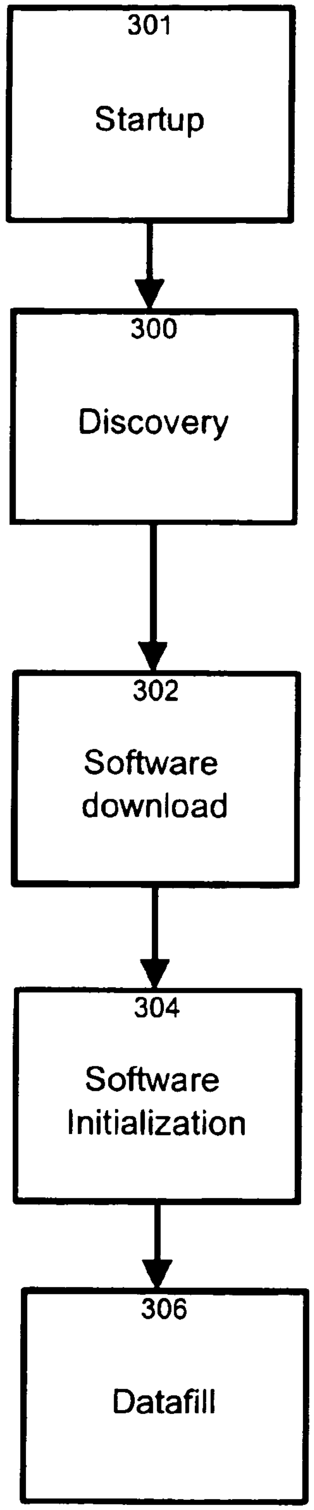

301
Startup
300
Discovery
302
Software download
304
Software Initialization
306
Datafill

Fig. 3

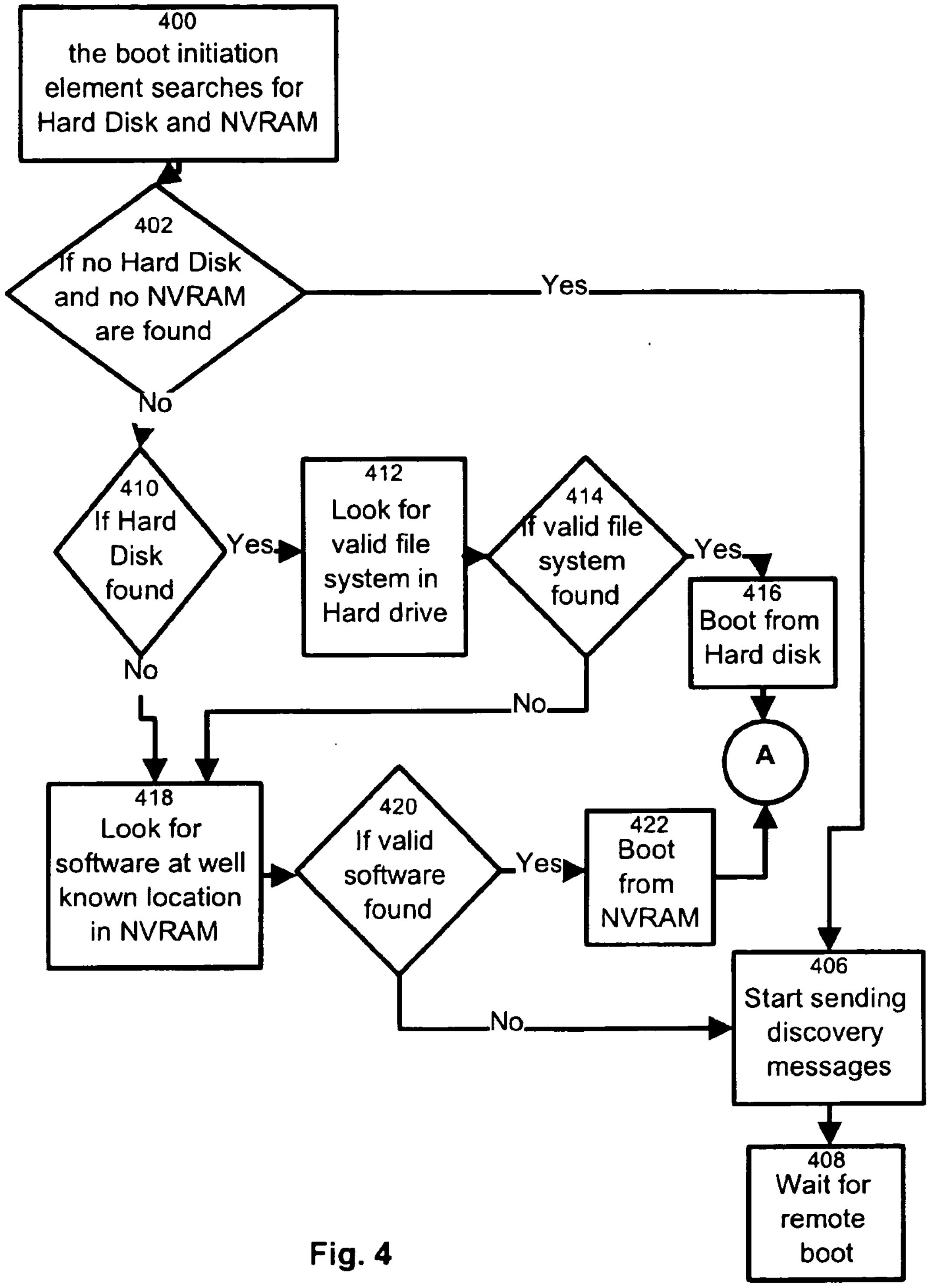

400
the boot initiation element searches for Hard Disk and NVRAM
402
If no Hard Disk and no NVRAM are found
Yes
No
410
If Hard Disk found
Yes
No
412
Look for valid file system in Hard drive
414
If valid file system found
Yes
No
416
Boot from Hard disk
A
418
Look for software at well known location in NVRAM
420
If valid software found
Yes
No
422
Boot from NVRAM
406
Start sending discovery messages
408
Wait for remote boot

Fig. 4

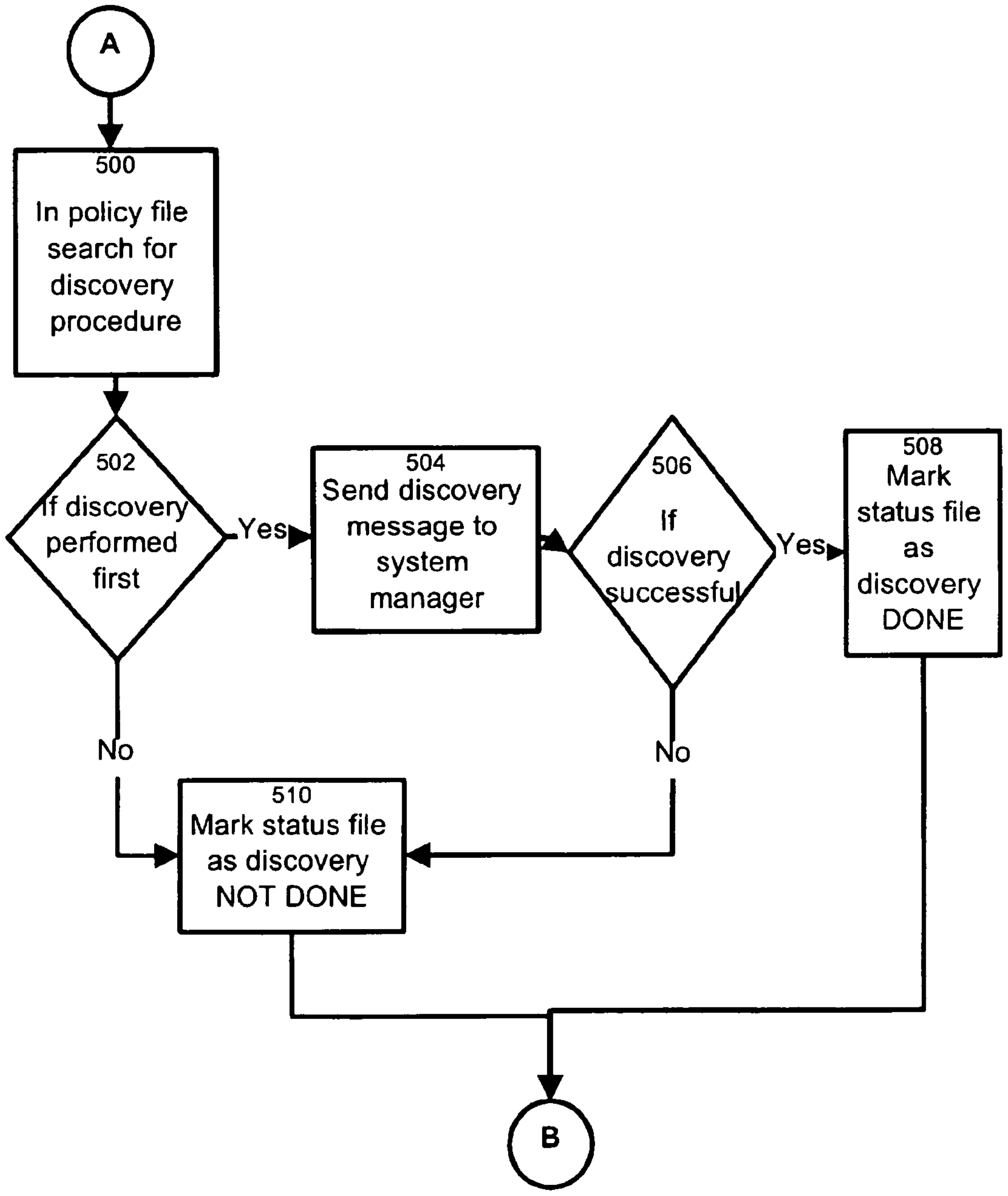

A
500
In policy file search for discovery procedure
502
If discovery performed first
Yes
No
504
Send discovery message to system manager
506
If discovery successful
Yes
No
508
Mark status file as discovery DONE
510
Mark status file as discovery NOT DONE
B

Fig. 5

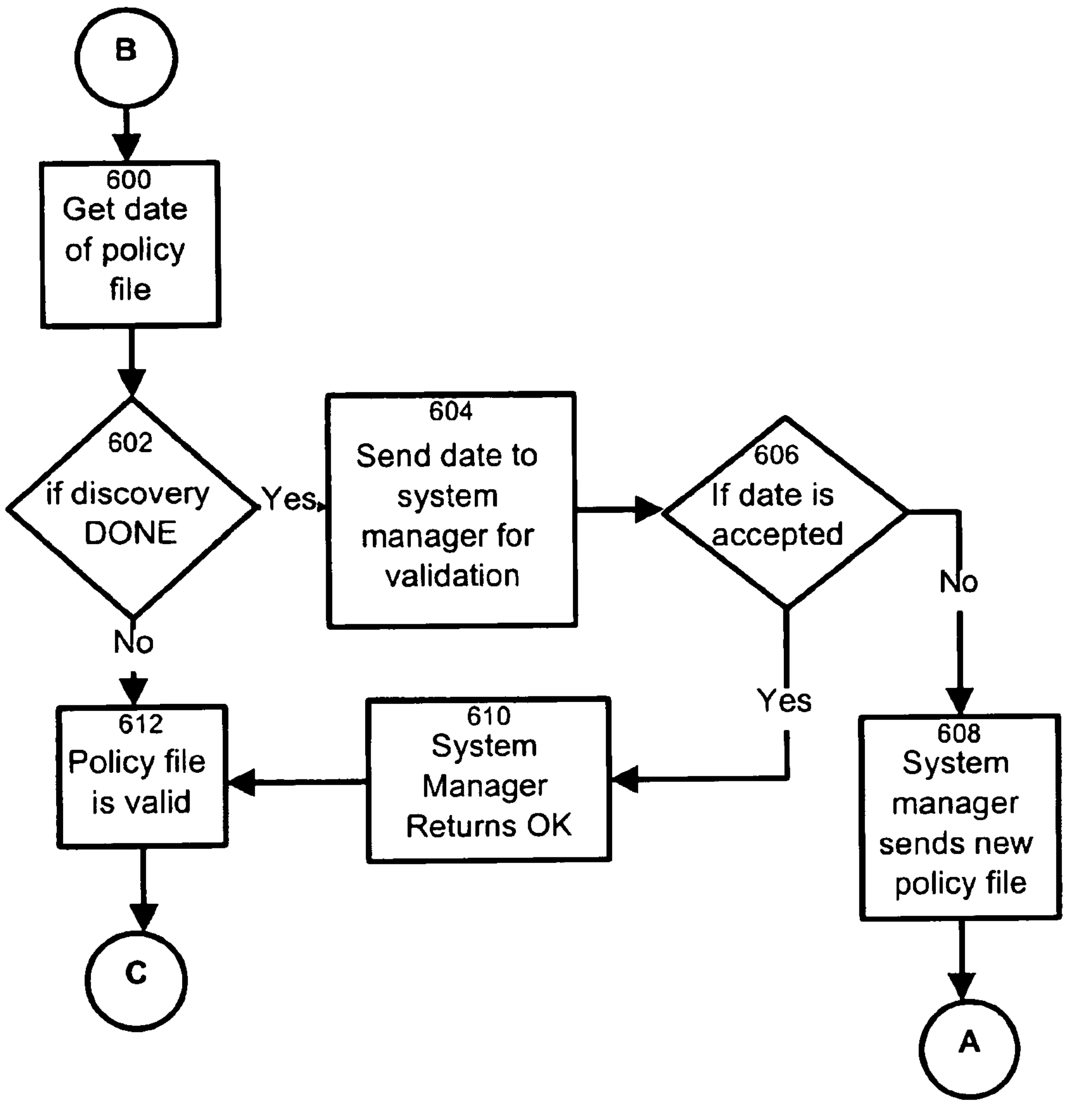

B
600
Get date of policy file
602
if discovery DONE
Yes
No
604
Send date to system manager for validation
606
If date is accepted
No
Yes
608
System manager sends new policy file
610
System Manager Returns OK
612
Policy file is valid
C
A

Fig. 6

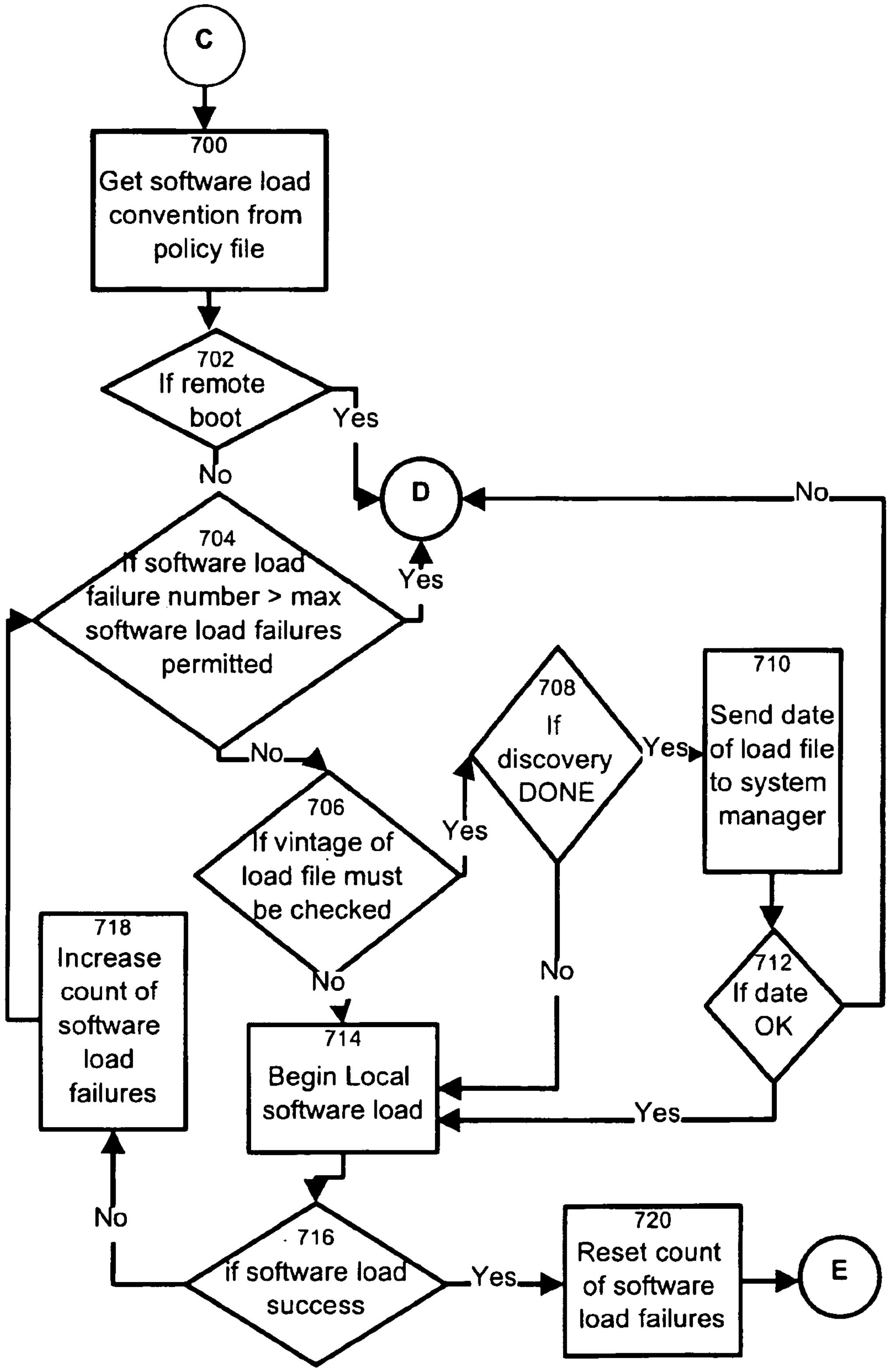

C
700
Get software load convention from policy file
702
If remote boot
Yes
No
D
No
704
If software load failure number > max software load failures permitted
Yes
No
708
If discovery DONE
Yes
710
Send date of load file to system manager
706
If vintage of load file must be checked
Yes
No
718
Increase count of software load failures
No
712
If date OK
714
Begin Local software load
Yes
No
716
If software load success
Yes
720
Reset count of software load failures
E

Fig. 7

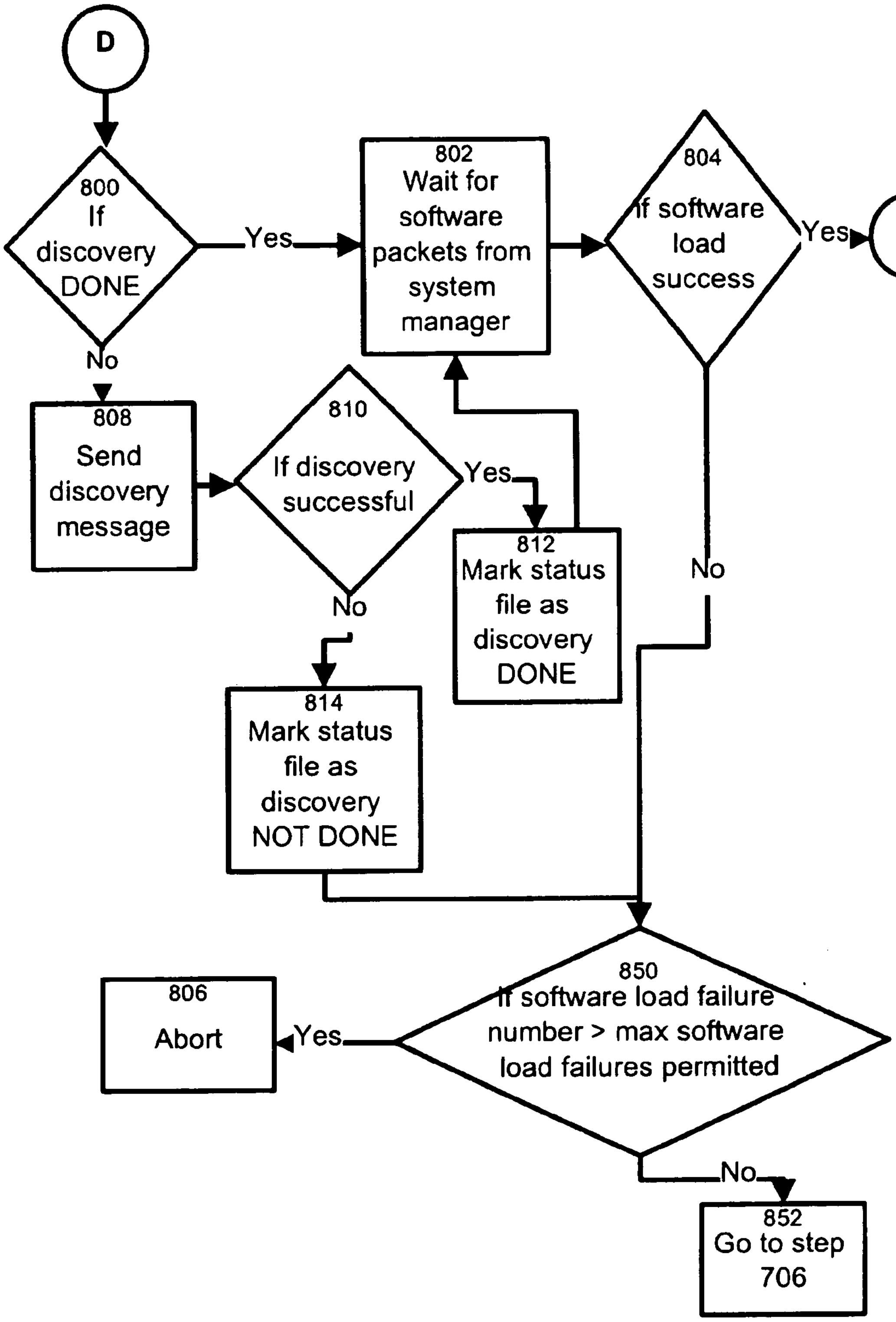

D
800
If discovery DONE
Yes
No
802
Wait for software packets from system manager
804
If software load success
Yes
E
No
808
Send discovery message
810
If discovery successful
Yes
No
812
Mark status file as discovery DONE
814
Mark status file as discovery NOT DONE
850
If software load failure number > max software load failures permitted
Yes
806
Abort
No
852
Go to step 706

Fig. 8

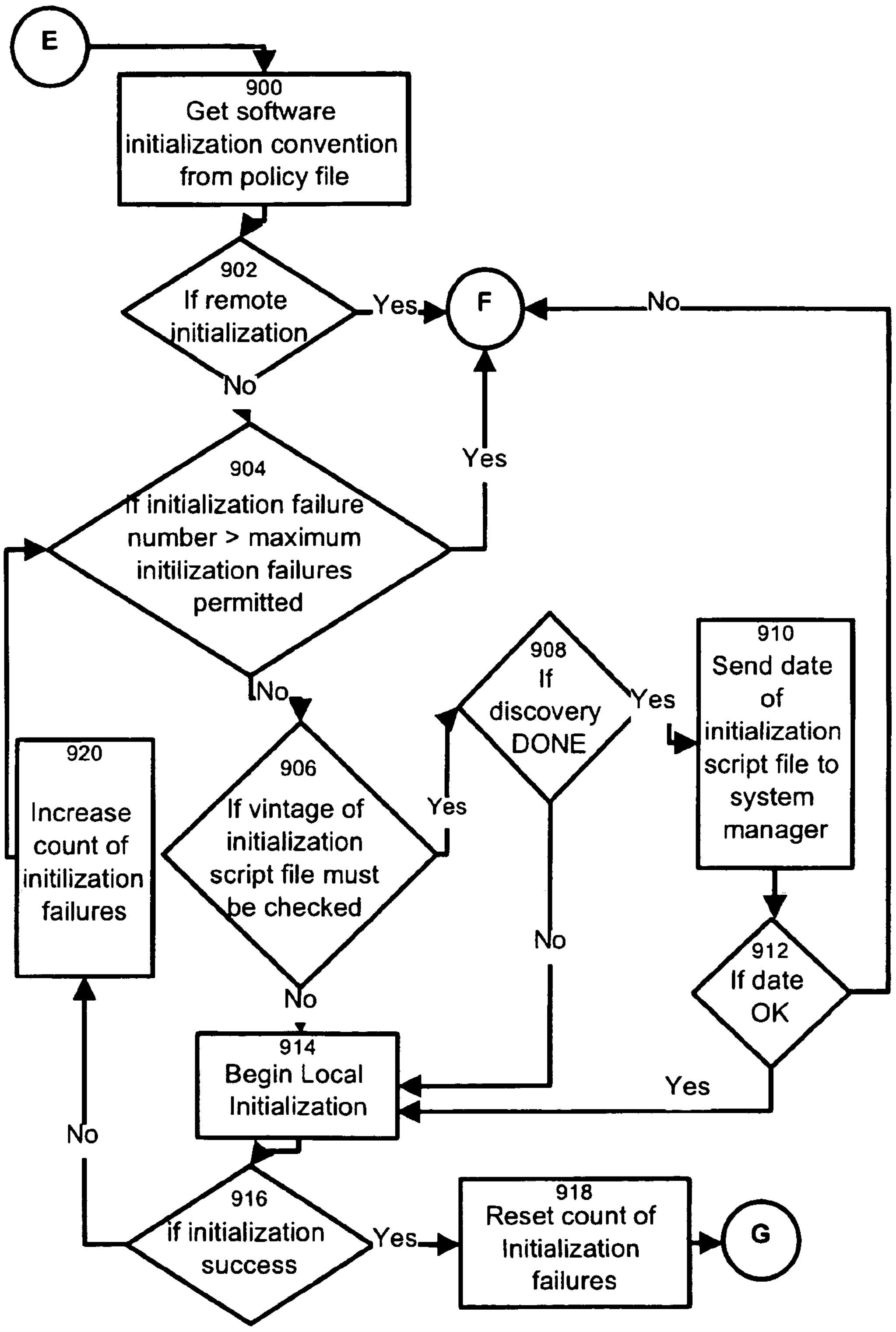

E
900
Get software initialization convention from policy file
902
If remote initialization
Yes
No
F
No
904
If initialization failure number > maximum initilization failures permitted
Yes
No
906
If vintage of initialization script file must be checked
Yes
No
908
If discovery DONE
Yes
No
910
Send date of initialization script file to system manager
912
If date OK
Yes
No
914
Begin Local Initialization
916
if initialization success
Yes
No
918
Reset count of Initialization failures
G
920
Increase count of initilization failures

Fig. 9

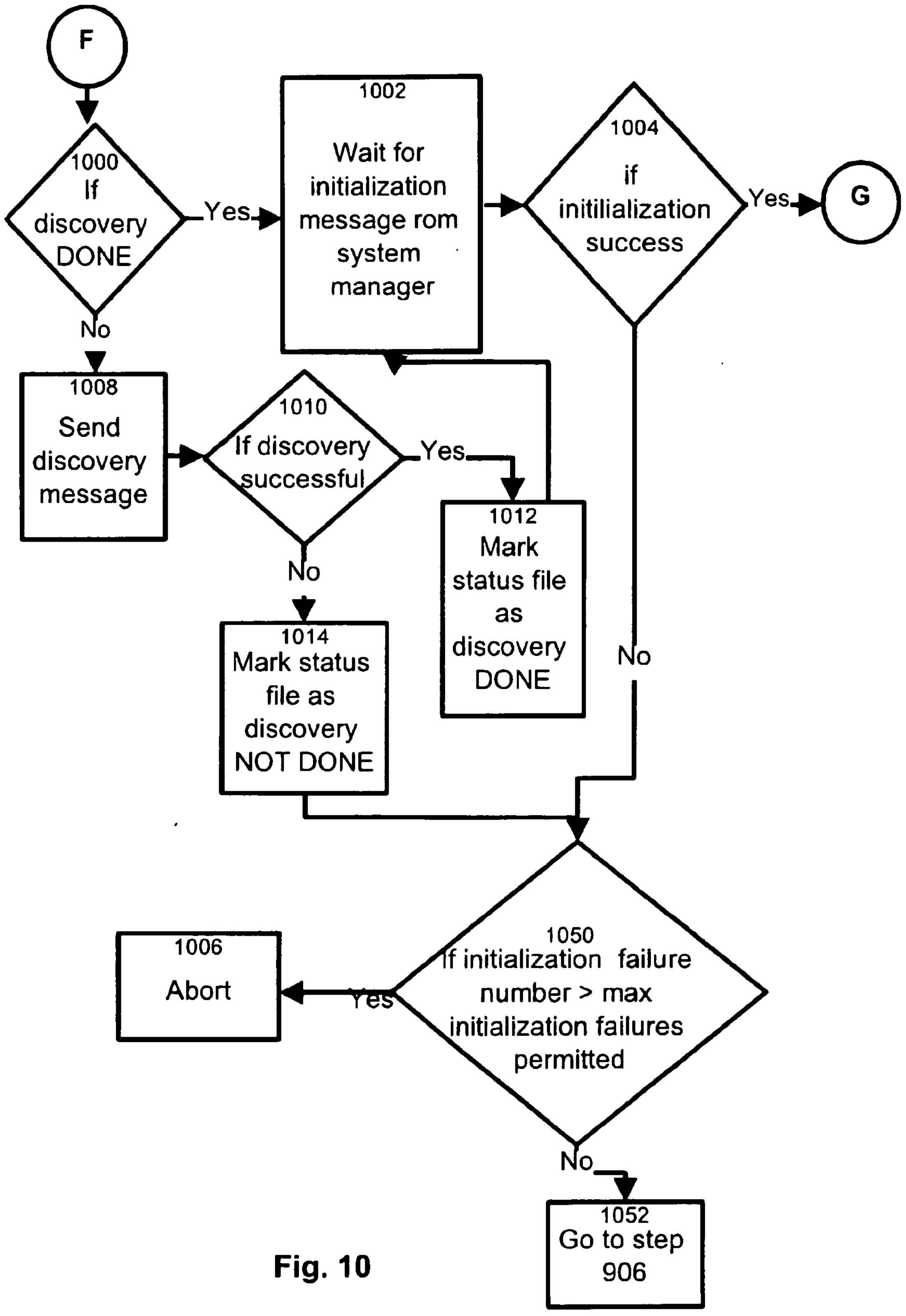

F
1000
If discovery DONE
Yes
No
1002
Wait for initialization message rom system manager
1004
if initilialization success
Yes
G
1008
Send discovery message
1010
If discovery successful
Yes
No
1012
Mark status file as discovery DONE
1014
Mark status file as discovery NOT DONE
No
1050
If initialization failure number > max initialization failures permitted
Yes
1006
Abort
No
1052
Go to step 906

Fig. 10

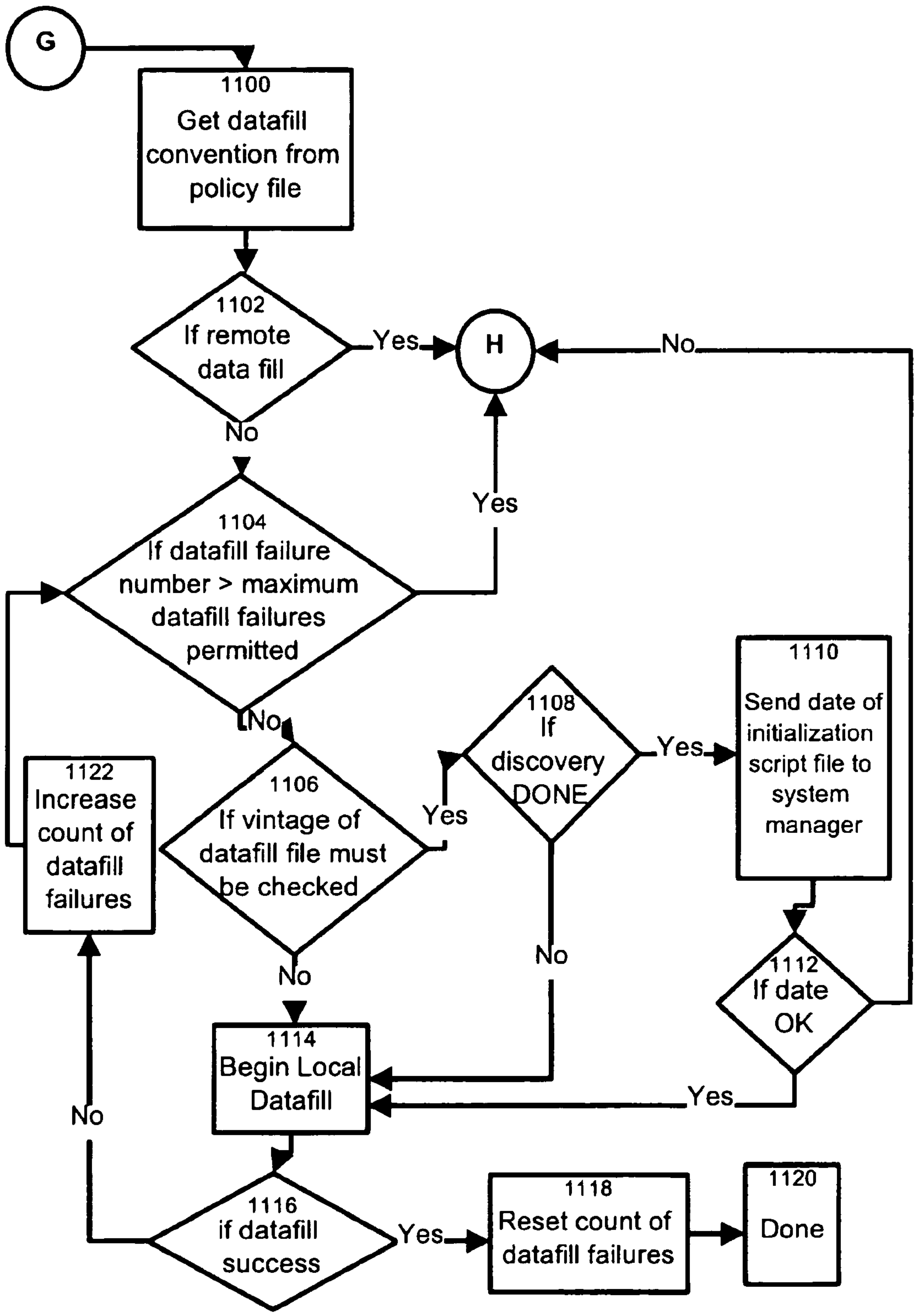

G
1100
Get datafill convention from policy file
1102
If remote data fill
Yes
H
No
1104
If datafill failure number > maximum datafill failures permitted
Yes
No
1106
If vintage of datafill file must be checked
Yes
No
1108
If discovery DONE
Yes
No
1110
Send date of initialization script file to system manager
1112
If date OK
Yes
No
1122
Increase count of datafill failures
1114
Begin Local Datafill
1116
if datafill success
Yes
No
1118
Reset count of datafill failures
1120
Done

Fig. 11

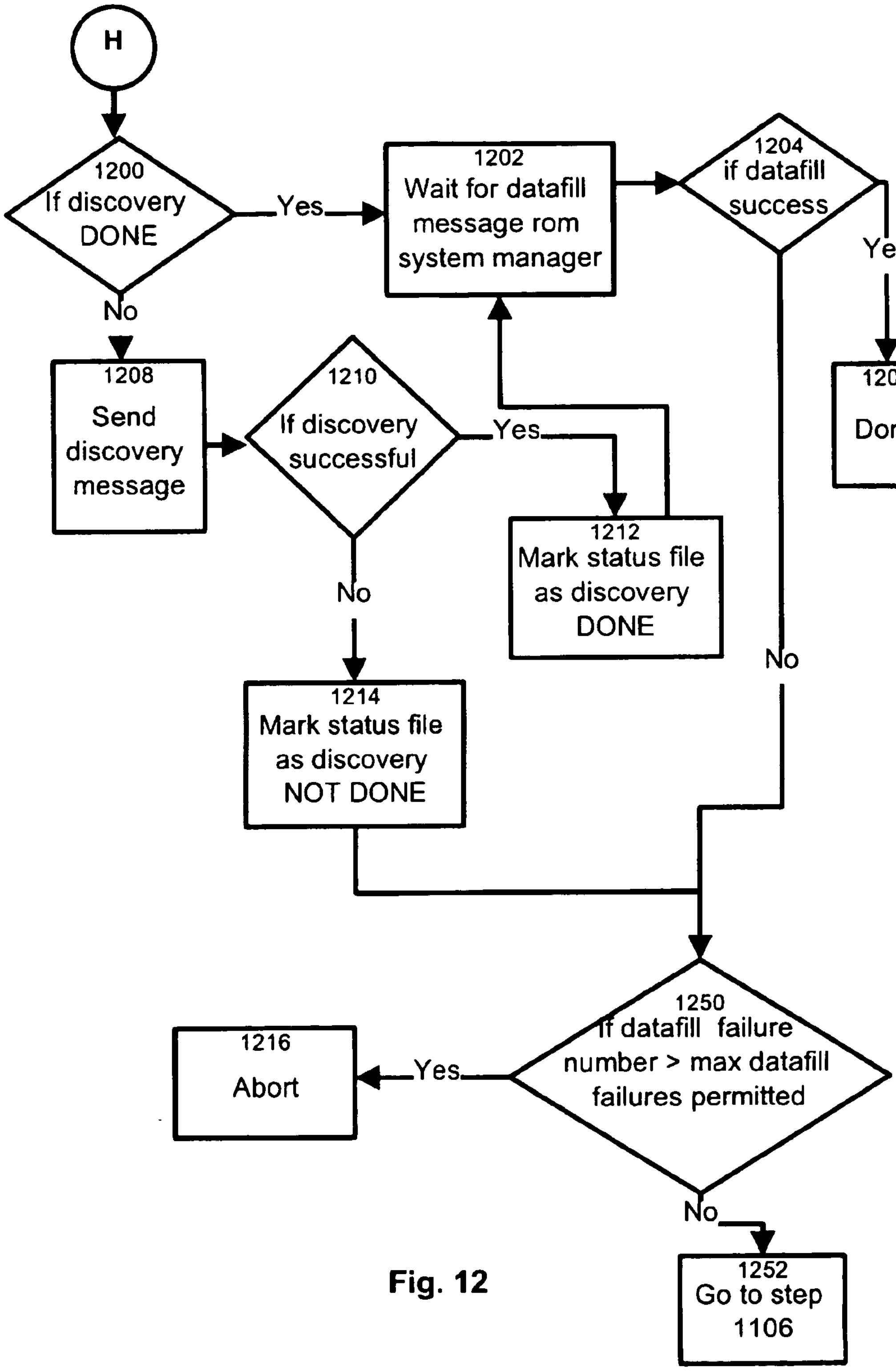

H
1200
If discovery DONE
Yes
No
1202
Wait for datafill message rom system manager
1204
if datafill success
Yes
No
1206
Done
1208
Send discovery message
1210
If discovery successful
Yes
No
1212
Mark status file as discovery DONE
1214
Mark status file as discovery NOT DONE
1250
If datafill failure number > max datafill failures permitted
Yes
No
1216
Abort
1252
Go to step 1106

Fig. 12

PROCESS AND APPARATUS FOR INITIALIZING A COMPUTER FROM POWER UP

FIELD OF THE INVENTION

This invention relates to a process and apparatus for starting up a computer system. It is applicable to computers operating in a distributed network environment and may be used to allow the individual computers to boot selectively from a local memory unit or from a network through the use of a policy file. The invention also provides a computer readable storage medium containing a program element to direct a computer to boot-up.

BACKGROUND

Booting is the process by which software (usually the operating system) is loaded into the memory of a computer and begins execution. Booting may also include loading a software image and starting software instances such as accounting or mail daemons. Usually, booting is done when the computer is turned on. Different computer systems can boot in different fashions. For example, most PCs are able to boot their operating system from a disk containing the required booting software and many embedded systems are able to boot from a pre-configured boot ROM. One type of network organization method involves using diskless workstations attached to a server that contains software programs and data. Diskless workstations are able to boot from a network using a connection protocol such as the TCP/IP protocol suite and download the programs in order to run them remotely. In these systems the elements of the TCP/IP protocol suite must also be fetched from a remote source and the booting is usually done from an IP network.

In a typical system, when a computer is first turned on, code present in a boot ROM is executed. Typically this code directs the computer to check for hardware components to ensure no essential components are lacking in the system. The boot ROM code then proceeds in loading up the operating system software. In the majority of computer systems, the booting procedure is hard-coded. For instance, a computer that boots from its hard disk always boots from its hard disk and one that boots from a network always boots from a network. Modifying the booting procedure involves reprogramming the boot ROM. Therefore if the hard drive of computer that boots from its hard drive fails, the computer cannot start although the same software could be available from the network. The reverse is also true for a computer that usually boots from the network when the latter fails.

Thus, there exists a need in the industry to provide an improved method for booting a computer such as to allow greater flexibility and to permit a computer to selectively boot from either its non-volatile memory unit or from a remote location, such as from a network.

OBJECTS AND STATEMENT OF THE INVENTION

An object of the invention is to provide a machine-readable storage medium containing an improved program element to boot-up a computer.

An object of this invention is to provide a novel method for booting-up a computer.

Yet another object of the invention is to provide a computer implementing a novel boot-up procedure.

As embodied and broadly described herein the invention provides a machine-readable storage medium containing a program element for directing a computer to effect a multi-stage boot-up procedure from power-up, said multi-stage boot-up procedure including at least two successive stages, said program element implementing functional blocks, including:

- a boot element for initiating one stage of said boot-up procedure;
- a data structure including a pointer to a location containing an information element, said boot element capable of interacting with said information element to cause execution of a stage of said boot-up procedure following in order said one stage, said pointer being selectively settable to point to either one of a local site and a remote site.

For the purpose of this specification, the expression "local site" refers to a location that is in the computer itself or, generally speaking, part of the computer. For example, a disk drive either internal or external is considered local because it is integrated to the structure of the computer. On the other hand, "remote site" is considered a location holding components that are not normally part of the computer, but with which the computer may be able to communicate. For example, a node in a network to which the computer is connected is considered to be "remote" because the node is normally a separate device from the computer, although the computer can communicate with that node through a predetermined protocol to exchange data.

For the purpose of this specification, the expression "vintage of a file" or "vintage" is used to designate the age or time characteristics of a file that indicates whether the file is up to date or not.

For the purpose of this specification, the expression "non-volatile storage" is used to designate a storage unit that maintains its content even if the storage device has no power such as non-volatile RAM (NVRAM) or a hard-disk.

In accordance with the present invention, the machine-readable storage medium holds a program element implementing a boot-up procedure. That procedure is characterized as being a multi-stage process. A data structure containing a pointer that can be selectively set to point toward a local site or a remote site indicates the location of software or data components necessary to complete the execution of one or more of the boot-up procedure stages. In a specific example, this provides a flexible booting strategy allowing initiating the boot-up procedure locally while loading some software components from a remote site, such as a network. The components that one may select to load from the remote location can be those that may be the subject of repeated upgrades or revisions. This avoids the necessity of changing the boot-up program at each node of the network, as an upgrade at the server that delivers the particular component is sufficient.

In a most preferred embodiment, the boot-up procedure includes five stages, namely start-up, discovery, software download, software initialization and datafill. Several files contain instructions indicating to the boot-up program how to proceed, in particular, where data or software components necessary for the execution of the various stages are located. In addition, a file may also point to an alternate resource location should the primary site from which the data or software component is to be obtained is inoperative.

The start-up stage initiates the boot-up procedure. The CPU of the computer, upon power-up, begins executing from a boot initiation element a set instructions starting at a particular address of a non-volatile memory. The purpose of the boot initiation element is to run basic sanity checks to determine all the critical components of the computer that are present, run the necessary I/O drivers and communication protocols, etc. In other words, the boot initiation element activates software components and sets-up the hardware to allow other software elements necessary to complete the boot-up procedure to run properly. The boot initiation element interacts with a policy file containing a plurality of entries that determine specifically how the boot-up procedure will be completed. Typically, the policy file that can be either held on a non-volatile memory or be part of the boot initiation element includes one or more pointers specifying where a particular information element such as software or data entities are to be acquired. In a most preferred embodiment, the policy file specifies when the discovery stage is to be performed. In short, the discovery stage is a registration of the computer system with the network. This involves setting up a data exchange transaction with a given node in the network to indicate to that node that a particular computer is being boot up. A successful registration procedure indicates that reliable communication with the network is likely. Accordingly, software or data elements that may be needed to complete the boot-up procedure can be acquired from that source. Otherwise, if the registration procedure is not successful, the boot initiation element may then revert solely to local resources since the network is not accessible.

The next step of the boot-up process is the software download stage that involves downloading basic software from the network or locating that software in a local resource, as specified by the policy file. Typically, the software download process is a process whose purpose is to acquire the bulk of the operating system. The policy file specifies from where the files must me acquired. A software load may also be present to describe what files need to be acquired. Once this stage is completed, the necessary executable files permitting to activate the operating system will be present in the computer memory.

The software initialization step follows next. In essence, the purpose of this stage is to begin execution of the files acquired during the previous stage in a particular order specified by a software initialization file. At this stage, the policy file indicates from where the software initialization file is to be obtained. Once this stage is completed, the files of the operating system are active in memory and ready to execute their respective tasks.

The datafill is the last stage of the boot-up procedure and includes the introduction of configuration data into the software instances. Here, data elements necessary for the proper operation of the operating system files are fetched from a datafill file and dispatched to the intended recipients. As was the case from the previous stages, the policy file indicates from where the datafill file is to be obtained.

As embodied and broadly described herein, the invention also provides a method for effecting a boot-up procedure of a computer, said boot-up procedure including first and second successive stages, said method comprising the steps of:

a) providing a data structure including a pointer to a location containing an information element, said pointer being selectively settable to point to either one of a local site and a remote site;

b) initiating the first step of said boot-up procedure;

c) processing said data structure to determine a location of said information element;

d) accessing said information element at the location determined at step c;

e) processing said information element to cause execution of said second stage of said boot-up procedure.

As embodied and broadly described herein, the invention also provides a computing apparatus, comprising:

a processor;

a memory holding a program element for directing said processor to execute a multi-stage boot-up procedure of said computing apparatus, said multi-stage boot-up procedure including at least two successive stages, said program element implementing a boot element for initiating one stage of said boot-up procedure;

a data structure in said memory, said data structure including a pointer to a location containing an information element, said boot element capable of interacting with said information element to cause execution of a stage of said boot-up procedure following in order said one stage, said pointer being selectively settable to point to either one of a local site and a remote site.

As embodied and broadly described herein, the invention also provides a machine-readable storage medium containing a program element for directing a computer to effect a boot-up procedure from power-up, said program element implementing a boot element for initiating said boot-up procedure, said boot element capable of interpreting data stored in a certain file during execution of said boot-up procedure, said boot element providing means to establish a data exchange transaction between the computer and a remote site to verify a vintage of said file.

Under an example embodying this particular aspect of this invention, the boot initiation element is capable of verifying the vintage of any one of the files that direct the execution of the boot-up procedure, namely the policy file, the software download file, the software initialization file and the datafill file. In short, the verification procedure involves the steps of communicating any suitable characteristic of the file such as its date of creation or its date at which it was last modified to a remote reference site, such as a node in the network. If the file is up to date, the reference site notifies the boot initiation element accordingly, and the latter utilizes the file contents to complete the execution of the boot-up procedure. On the other hand, if the file is no longer up to date, and a new version exists, the reference center then transmits the new file to the boot initiation element that replaces the old file by the new one.

As embodied and broadly described herein the invention also provides a method for effecting a boot-up procedure of a computer, said method comprising the steps of:

a) providing a file in said computer containing data elements, said data elements influencing execution of said boot-up procedure;

b) initiating a data exchange transaction with a remote site to determine a vintage of said file; and

c) completing said boot-up procedure.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other features of the present invention will become apparent from the following detailed description considered in connection with the accompanying drawings. It is to be understood, however, that the drawings are designed for purposes of illustration only and not as a definition of the limits of the invention for which reference should be made to the appending claims.

FIG. 1 shows a distributed computing network where the process in accordance with this invention can be implemented;

FIG. 2 shows a computing node consistent with the spirit of the invention connected via an ATM link to a system manager unit;

FIG. 3 shows five stages of the booting process in accordance with the spirit of the invention;

FIG. 4 shows flow-charts of the startup stage in accordance with the spirit of the invention;

FIG. 5 & 6 shows flow-charts of the discovery stage in accordance with the spirit of the invention;

FIG. 7 & 8 show flow-charts of the software download stage in accordance with the spirit of the invention;

FIG. 9 & 10 show flow-charts of the software initialization stage in accordance with the spirit of the invention;

FIG. 11 & 12 show flow-charts of the datafill stage in accordance with the spirit of the invention;

DESCRIPTION OF A PREFERRED EMBODIMENT

The present invention is concerned with a process that allows a computer to boot selectively from its own local non-volatile memory unit or from a network. In the preferred embodiment of this invention, when a local non-volatile memory unit is available, the system uses a policy file located in the non-volatile memory unit. The policy file directs each stage of the booting process independently, which allows for greater flexibility during the booting process. Preferably, the policy file is easily modified to reflect the system changing requirements. Furthermore, failure recovery mechanisms allow the computer to revert to an alternative booting scheme. For example, in the case where a local non-volatile memory unit is in failure or not available, the computer can wait for the system manager to send the booting instructions through the network. These features and others will become apparent following the description in the sections that follow.

Computers that may make use of the booting method in accordance with this invention may be used in stand-alone mode or be of a distributed nature, as shown in FIG. 1. These machines, herein designated as nodes, may reside in geographically remote locations and communicate using a set of predefined protocols. Protocols such as TCP/IP, client/server architecture and message passing are all possible methods of achieving a distributed computing environment. For more information on distributed processing, the reader is invited to consult Operating Systems by William Stallings, Prentice Hall 2nd edition 1995. The text of this document is included hereby by reference.

In the preferred embodiment of this invention, as illustrated in FIG. 2, the computing node to be booted 201, herein referred to as the new computing node, is in a computer network comprising a system manager entity 200. Preferably, the new computing node 201 and the node including the system manager 203 communicate through an ATM link or other network connection. The new computing node comprises a boot initiation element software module 202 and may include a hard disk or a non-volatile RAM (NVRAM) unit 206.

The boot initiation element 202 is the first piece of code that is executed when the computer powers on. Preferably, the code begins at an address of the memory that is the default address loaded in the code segment of the CPU upon power-up. Thus, when the CPU is energized and becomes operational it begins executing the instructions of the boot initiation element. The boot initiation element includes software allowing the computer to perform a network boot namely an ATM driver allowing it to communicate with the system manager, software to run basic sanity checks on the system, software providing a simple communication protocol and software to download information from the network

such as Trivial File Transfer Protocol (TFTP). Furthermore, the boot initiation element includes instructions to check for a hard drive and a non-volatile RAM module (NVRAM) 206 and code to interpret the file system 208. Preferably, the same algorithm is used whether the computer node is a disk-based or diskless computer.

The file system 208 comprises a series of files used in the booting process as well as pointers to these files to allow the boot initiation element 202 to access them. In the preferred embodiment, the file system includes a policy file, a status file, an initialization script file, a datafill file and software load files. Each of these files and their purpose will be described in detail later in this specification. The structure of the file system may vary across implementations and variations in the number of files or in the structure of the file system do not detract from the spirit of the invention.

In the most preferred embodiment of this invention, the booting process is divided into five stages, as illustrated in FIG. 3, namely start-up 301, discovery 300, software download 302, software initialization 304 and datafill 306. Each stage may be driven either from the local hard drive (or NVRAM) or from the system manager on the network depending on the instructions in a policy file.

In a preferred embodiment the policy file is a file comprising a set of entries that control the booting process. It is located in the hard drive or other non-volatile memory unit and is accessible via the filing system 208. The main advantage of having the policy file in the hard drive is that the policy file may be easily re-configured by the network manager according to the changing requirements of the network. Alternatively, the policy file may be part of the boot initiation element 202. A role of the policy file is to specify if software or data entities are to be acquired from the remote system manager through the network or from a local memory unit such as a hard drive or an NVRAM unit. Before each stage in the booting process shown in FIG. 3, the policy file is examined in order to determine if the stage must be run locally or remotely.

The first stage in FIG. 3, the start-up stage 301, involves the CPU of the computer upon power-up to begin executing a boot initiation element. The instructions in the boot initiation element are performed starting at a particular address of a non-volatile memory unit such as a ROM unit or other suitable devices. The purpose of the boot initiation element is to run basic sanity checks to determine all the critical components of the computer that are present, run the necessary I/O drivers and communication protocols, etc. In other words, the boot initiation element activates software components and sets-up the hardware to allow other software elements necessary to complete the boot-up procedure to run properly. In a typical flow of events, as shown in FIG. 4, when the power is first initialized in the computer, the boot initiation element code begins to execute and searches for a hard drive and a non-volatile memory unit 400. Condition 402 is checked, and if neither a hard drive nor a non-volatile memory unit are found, the boot initiation element using the ATM driver code initiates the discovery process 406 by sending a registration message to the system manager. Following this, the computer waits for a remote boot 408 and for software to be received from the network. If condition 402 is answered in the negative, condition 410 is checked. If a hard disk is found then it is searched 412 for a file system containing the files required for booting. Condition 414 is answered in the positive if a valid file system is found and the computer will boot from the hard disk 416. If conditions 410 or 414 are answered in the negative, namely if there is no hard disk or if the hard disk does not contain a valid file system, the NVRAM is searched for the software required 418 located at a well-known location. If the software is not found as checked by condition 420, the boot initiation element, using the ATM driver code, initiates the discovery process 406 by sending a registration message to the system manager. Following this, the computer waits for a remote boot 408 and for software to be received from the network. However, if condition 420 is answered in the positive, the computer will boot from the NVRAM 422. Once the source of the booting process has been selected, booting can begin as shown in FIG. 5.

The following stage, the discovery stage 300, involves the new computer registering with the system manager through the network. Essentially, using the ATM driver, the new computer sends messages to the system manager identifying its presence and its location. The system manager typically registers the new computer node and sends an acknowledgement message. The discovery stage can be performed at any time during the booting process. For example, if the software download 302 and initialization stage 304 are to be performed locally (from the hard drive or from the NVRAM) and the datafill stage 306 needs to be done from the network, the discovery may be diferred until the datafill stage is ready to begin since the system manager doesn't need to know where the new computer is located until it must send data or messages to it. In a typical flow of events, as shown in FIG. 5, the interpreter in the boot initiation element examines the policy file 500 and determines if discovery must take place immediately. In the preferred embodiment of this invention, the policy file contains instructions of the form:

```
Discovery{BEFOREDISKBOOT | AFTERDISKBOOT | AFTERINITSCRIPT |
        AFTERDATAFILL}
Policy File Vintage{CHECK|NO CHECK}
```

where the vertical bars indicate that only one of the options needs to be specified. If condition 502 is answered in the positive, the discovery is to be performed immediately. The boot initiation element then sends a discovery message 504 to the system manager. Condition 506 verifies if an acknowledgement message is received from the system manager indicating a successful discovery. If condition 506 is answered in the affirmative, an entry is made to a status file 508 indicating that the discovery is done. If either condition 502 or condition 506 is answered in the negative, an entry is made to a status file 510 indicating that the discovery is not done. The status file is used to maintain status information about the booting process including which stages have been performed and whether or not they were successful. The network manager can examine this file in order to obtain a "footprint" of the cause of an unexpected failure. It can also be used to verify that all the stages in the booting process were executed as specified in the policy file. In essence, the status file is there to keep a log of the booting process and is not critical in the booting process. Therefore the absence of a status file in an embodiment of this invention does not detract from the spirit of this invention. In addition to indicating if a booting stage was successful, entries in the status file may also include the number of time a certain stage has failed and whether the vintage of a given file has been verified. After steps 510 and 508 we proceed in validating the policy file as shown in FIG. 6. Preferably, every time a new file is accessed in order to obtain booting

instructions, a verification operation is performed, herein referred to as checking the vintage of the file, in order to determine if the file contains the current version of the instructions. The general idea in the validation involves comparing the date of the policy file in the new computer with the date of the policy file in the system manager. If the file being verified is up to date, the system manager sends an approval message, otherwise the system manager sends an up-to-date version of the file. The first step in checking the vintage of the policy file involves extracting the date of the policy file in the new computer 600. If the discovery has been done, condition 602 is answered in the positive and the new computer sends a message to the system manager containing the date of the policy file 604. If the date of the policy file is valid, condition 606 is answered in the positive and the system manager sends a message to proceed 610, and the new computer then considers the policy file valid 612. If the discovery has not been done, condition 602 is answered in the negative and the policy file is also assumed to be valid 612. If the date of the policy file is not valid, condition 606 is answered in the negative and the system manager sends a message to the new computer along with a new policy file 608. In the case where a new policy file is sent, the system starts over (tag A) by examining the discovery convention. In the case where the policy file was valid, the system proceeds to the software download stage 302 (tag C).

The software download stage 302 involves downloading basic software from the network. In the case where the system has a hard drive and is to obtain the software locally, this stage may involve locating the software in the local storage device, with the use of a load description file identifying the software image on the computer. Preferably the software downloaded includes software to download yet more software. The software image may include the basic operating system and a node management module. In the preferred embodiment the software load is a single executable file. Other types of files may constitute a software load without detracting from the spirit of the invention. The software load may also include a load description file that describes which software runs on a given node. Preferably, the load description file is small so that it may be easily transferred to the system manager for a vintage check. Optionally, more than one load image may be present in the software load files to accommodate operating systems with multiple address space such as UNIX. For these operating systems, separate software load files may be started in different address spaces. In a typical interaction as shown in FIG. 7, the load convention is obtained from the policy file 700 that indicates if the software load is to be obtained remotely or locally. Preferably, computers with local hard-drives or NVRAMs boot locally in order to improve the speed of the booting process and the system manager only participates in the process if needed. If the software load is to be obtained locally, then condition 702 is answered in the negative. Preferably, a failure recovery mechanism is present which allows the computer to boot from an alternative memory source in the case where the preferred source designated in the policy file is unavailable. For example, if the policy file requires software or data to be obtained from a remote system manager and the latter is not accessible for whatever reason, the computer will boot from the local hard drive or NVRAM if the required software or data is valid and present. The converse is also possible, namely if the policy file requires software or data to be obtained from the local hard drive and the latter is not accessible for whatever reason, the computer will boot from the system manager if the required software or data is valid and present. In the case where both the hard drive (or NVRAM) and remote system manager are unavailable, the computer will not boot. Therefore, the computer keeps track of the number of failures. If the number of failures is below a pre-determined allowed number of failures, condition **704** is answered in the negative and the software load will be obtained from the hard disk or NVRAM. Preferably, the required information about the software load is located in a load description file that indicates which files are in the software load and what is their vintage. The policy file is then examined to determine if the vintage of the load file must be verified before proceeding. If the vintage of the load files must not be checked, condition **706** is answered in the negative and the computer proceeds in booting from the local hard-drive (or NVRAM) **714**. Booting from the hard drive may include searching for predetermined files. Condition **716** verifies if the booting was a success. If the boot was a success, step **720** resets the number of boot failures to zero and proceeds to the software initialization stage (tag E). If the boot wasn't a success, condition **716** is answered in the negative and the number of failures is incremented **718**. Condition **704** is then re-evaluated. If the vintage of the load file must be checked, condition **706** is answered in the positive and the computer proceeds in verifying the date of the load file. If the discovery stage has already been done, condition **708** is answered in the positive and the date of the load file is sent to the system manager **710**. At condition **712** the system manager evaluates if the load file for the computer is the most up-to-date file. If the date of the load file is accepted, condition **712** is answered in the positive and the local boot can begin at step **714**. If at condition **708** the discovery stage has not taken place, the condition is answered in the negative and the load file is assumed to be valid and allows the local boot to begin at step **714**. The node proceeds in booting from the network (tag D) in three cases. The first case occurs if the policy file indicates that the software load must be obtained from the network causing condition **702** to be answered in the positive. The second case occurs if the local booting has failed more than a pre-determined number of times causing condition **704** to be answered in the positive. And finally, the third case occurs if the system manager did not accept the date of the load file causing condition **712** to be answered in the negative. The network booting procedure is shown in the flow chart of FIG. **8**. Condition **800** is answered in the positive if the status file indicates that the discovery has been done. The computer then waits for the system manager to send packets **802** containing the software load. If the software load is a success, condition **804** is answered in the positive, the status file reflects that the software load stage was successful and the computer proceeds to the software initialization stage (tag E). However, the software load stage may fail for a number of reasons. For example, the system manager may be unavailable, the packets may have gotten lost, or the connection may have been broken. If the software load stage fails, condition **804** is answered in the negative. The computer verifies if the number of failures exceeds the number of failures permitted. If it does, condition **850** is answered in the positive and the process is aborted **806**. However, if the number of failures is less than the maximum

number permitted, condition **850** is answered in the negative and the computer attempts a local software load stage **852** starting at stage **706** on FIG. **7**. If at condition **800** the discovery has not been done, the computer begins the discovery process by sending discovery messages **808** across the network to the system manager. If the discovery is successful, condition **810** is answered in the positive and an entry is made to the status file to indicate that the discovery has been done. The computer then waits for the system manager to send packets **802** containing file load information. In the event that the discovery fails, condition **810** is answered in the negative and an entry is made to the status file to indicate that the discovery was not done **814**. The process then proceeds to condition **850**.

The following stage, the software initialization stage **304**, involves starting and initializing part of the software received or located in the software download stage **302**. Software initialization involves executing code, preferably a script file, which instructs the computer to start a set of programs or processes. The preferred embodiment of this invention uses an initialization script file. This file includes a set of commands to start or create software instances on the computer. In its simplest form, the initialization script file is a sequence of start() and create() messages sent to different components of the software load. In a typical interaction as shown in FIG. **9**, the software initialization convention is obtained from the policy file **900** that indicates if the initialization script file is to be obtained remotely or locally. If the initialization script is to be obtained locally, then condition **902** is answered in the negative. The computer then checks how many times the initialization procedure has failed. If the number of failures is below a pre-determined allowed number of failures, condition **904** is answered in the negative and the initialization script will be obtained from the hard disk or NVRAM. The policy file is then examined to determine if the vintage of the initialization script file must be verified before proceeding. If the vintage of the initialization script file does not need to be checked, condition **906** is answered in the negative and the computer proceeds in executing the initialization script from the local hard-drive (or NVRAM) **914**. Condition **916** verifies if the initialization was a success. If the boot was a success, step **918** resets the number of initialization failures to zero and proceeds to the datafill stage (tag G). If the initialization was not a success, condition **916** is answered in the negative and the number of failures is incremented **920**. Condition **904** is then re-evaluated. Returning to condition **906**, if the vintage of the initialization script file must be checked, condition **906** is answered in the positive and the computer proceeds in verifying the date of the file. If the discovery stage has already been done, condition **908** is answered in the positive and the date of the initialization script file is sent to the system manager **910**. At condition **912** the system manager evaluates if the initialization script file for the computer is the most up-to-date file. If the date of the initialization script file is accepted, condition **912** is answered in the positive and the local initialization can begin at step **914**. If at condition **908** the discovery stage has not taken place, the condition is answered in the negative and the initialization script file is assumed to be valid and allows the local initialization to begin at step **914**. The node proceeds in initializing from the network (tag F) in three cases. The first case occurs if the policy file indicates that the initialization script file must be obtained from the network causing condition **902** to be answered in the positive. The second case occurs if the local initialization has failed more than a pre-determined number of times causing condition **904** to be answered in the positive. The third case occurs if the system manager did not accept the date of the initialization script file causing condition 912 to be answered in the negative. The network initialization procedure is shown in the flow chart of FIG. 10. Condition 1000 is answered in the positive if the status file indicates that the discovery has been done. The computer then waits for the system manager to send packets 1002 containing the initialization script and then executes the script. If the initialization is a success, condition 1004 is answered in the positive and the computer proceeds to the datafill stage (tag G). However, the initialization may fail for a number of reasons. For example, the system manager may be unavailable, the packets may have gotten lost, or the connection may have been broken. If the initialization fails, condition 1004 is answered in the negative. If the software load stage fails, condition 804 is answered in the negative. The computer then verifies if the number of failures exceeds the number of failures permitted. If it does, condition 1050 is answered in the positive and the process is aborted 1006. However, if the number of failures is less than the maximum number permitted, condition 1050 is answered in the negative and the computer attempts a local software initialization 1052 starting at stage 906 on FIG. 9. If at condition 1000 the discovery has not been done, the computer begins the discovery process by sending discovery messages 1008 across the network to the system manager. If the discovery is successful, condition 1010 is answered in the positive and an entry is made to the status file to indicate that the discovery has been done. The computer then waits for the system manager to send packets 1002 containing the initialization script and then executes the script. In the event that the discovery fails, condition 1010 is answered in the negative and an entry is made to the status file to indicate that the discovery was not done 1014. The process then proceeds to condition 1050.

Once the software instances have been started or created through the commands of the initialization script file, the datafill stage 306 involves loading the data into the software instances initialized by the software initialization stage 304. In the preferred embodiment, a datafill file contains a sequence of instructions, herein referred to as messages, which direct the computer as to how to start the software instances. In a typical interaction as shown in FIG. 11, the datafill convention is obtained from the policy file 1100 that indicates if the datafill file is to be obtained remotely or locally. If the datafill file is to be obtained locally, then condition 1102 is answered in the negative. The computer then checks how many times the datafill procedure has failed. If the number of failures is below a pre-determined allowed number of failures, condition 1104 is answered in the negative and the datafill file will be obtained from the hard disk or NVRAM. The policy file is then examined to determine if the vintage of the datafill file must be verified before proceeding. If the vintage of the datafill file does not need to be checked, condition 1106 is answered in the negative and the computer proceeds in executing the datafill file code from the local hard-drive (or NVRAM) 1114. Condition 1116 verifies if the datafill was a success. If the datafill was a success, step 1118 resets the number of datafill failures to zero and the computer initialization is done 1120 successfully. If the datafill was not a success, condition 1116 is answered in the negative and the number of failures is incremented 1122. Condition 1104 is then re-evaluated. Returning to condition 1106, if the vintage of the datafill file must be checked, condition 1106 is answered in the positive and the computer proceeds in verifying the date of the file. If the discovery stage has already been done, condition 1108

is answered in the positive and the date of the datafill file is sent to the system manager 1110. At condition 1112 the system manager evaluates if the datafill file for the computer is the most up-to-date file. If the date of the datafill file is accepted, condition 1112 is answered in the positive and the local datafill can begin at step 1114. If at condition 1108 the discovery stage has not taken place, the condition is answered in the negative and the datafill file is assumed to be valid, the system is allowed to begin local datafill at step 1114. The computer node obtains the datafill file from the network (tag H) in three cases. The first occurs if the policy file indicates that the datafill file must be obtained from the network causing condition 1102 to be answered in the positive. The second case occurs if the local datafill has failed more than a predetermined number of times causing condition 1104 to be answered in the positive. The third case occurs if the system manager did not accept the date of the datafill file causing condition 1112 to be answered in the negative. The network datafill procedure is shown in the flow chart of FIG. 12. Condition 1200 is answered in the positive if the status file indicates that the discovery has been done. The computer then waits for the system manager to send packets 1202 containing the datafill file and then executes the code. If the datafill is a success, condition 1204 is answered in the positive and the computer has finished booting. However, the remote datafill operation may fail for a number of reasons. For example, the system manager may be unavailable, the packets may have gotten lost or the connection may have been broken. If the datafill fails, condition 1204 is answered in the negative. The computer then verifies if the number of failures exceeds the number of failures permitted. If it does, condition 1250 is answered in the positive and the process is aborted 1216. However, if the number of failures is less than the maximum number permitted, condition 1250 is answered in the negative and the computer attempts a local datafill stage 1252 starting at stage 1106 on FIG. 11. If at condition 1200 the discovery has not been done, the computer begins the discovery process by sending discovery messages 1208 across the network to the system manager. If the discovery is successful, condition 1210 is answered in the positive and an entry is made to the status file to indicate that the discovery has been done. The computer then waits for the system manager to send packets 1202 containing the datafill file and then executes the code. In the event that the discovery fails, condition 1210 is answered in the negative and an entry is made to the status file to indicate that the discovery was not done 1214. The process then proceeds to condition 1250.

Although the present invention has been described in considerable detail with reference to certain preferred embodiments thereof, variations and refinements are possible without departing from the spirit of the invention. Therefore, the scope of the invention should be limited only by the appended claims and their equivalents.

What is claimed is:

1. A machine-readable storage medium containing a program element for directing a computer to effect a multi-stage boot-up procedure from power-up, said multi-stage boot-up procedure including a first stage and a plurality of successive stages following said first stage, the implementation of the stages subsequent to the first stage being effected by executing respective program code units, said program element including:

a boot initiation program module for execution by the computer to implement a first stage of said boot-up procedure;

a policy file operative for directing execution of stages subsequent to the first stage of said boot-up procedure, said policy file having a plurality of entries, each entry pointing to a location containing a program code unit suitable for execution by the computer to implement a respective stage of said multi-stage boot-up procedure, at least one entry pointing toward a location containing a first program code unit residing locally at the computer and at least another entry pointing toward a location containing a second program code unit residing at a site remote from the computer, the first program code unit being associated with a stage of the boot-up procedure that occurs subsequently to the first stage, the first program code unit residing locally at the computer prior to the execution of the first stage.

2. A machine-readable storage medium as defined in claim 1, wherein said boot initiation program module is capable of interacting with at least one of entry in said policy file.

3. A machine-readable storage medium as defined in claim 2, wherein said boot initiation program module is operative for searching locally at the computer for a memory unit.

4. A machine-readable storage medium as defined in claim 3, wherein said boot initiation program module is operative for searching the memory unit for program code units.

5. A machine-readable storage medium as defined in claim 2, wherein said boot initiation program module is operative for establishing a data exchange transaction between the computer and the site remote from the computer.

6. A machine-readable storage medium as defined in claim 5, wherein said policy file includes a first entry, said boot initiation program module being operative to process the first entry to issue a message to the site remote from the computer.

7. A machine-readable storage medium as defined in claim 6, wherein said policy file includes a second entry indicative of a location where program code units associated to a given stage reside.

8. A machine-readable storage medium as defined in claim 7, wherein said boot initiation program module is operative for processing said second entry to determine the location where the program code unit associated to a given stage resides.

9. A machine-readable storage medium as defined in claim 8, wherein upon occurrence of a failure of an attempt for processing the program code unit at a location pointed to by said second entry, said boot initiation program module is operative to process a program code unit at another location.

10. A machine-readable storage medium as defined in claim 9, wherein said second entry points locally at the computer and said another location is a site remote from the computer.

11. A machine-readable storage medium as defined in claim 9, wherein said second entry points toward a site remote from the computer and said another location is locally at the computer.

12. A machine-readable storage medium as defined in claim 5, wherein said boot initiation program module is operative for establishing a data exchange transaction to verify a vintage associated to said policy file, the vintage comprising at least one data element indicative of a time characteristic.

13. A machine-readable storage medium as defined in claim 2, wherein said policy file is a script file, said script file including a plurality of entries, at least one of said entries directing said boot initiation program module to execute a given program code unit.

14. A machine-readable storage medium as defined in claim 13, wherein said given program code unit is a software download file.

15. A machine-readable storage medium as defined in claim 13, wherein said given program code unit is a datafill file, sa datafill file including a plurality of entries, at least one of said en tries directing said computer to transfer data to a certain file other than said datafill file.

16. A method for effecting a boot-up procedure of a computer, said boot-up procedure including a first stage and a plurality of successive stages following said first stage, the stages subsequent to the first stage being effected by executing respective program code units, said method comprising the steps of:

a) providing a policy file having a plurality of entries, each entry pointing to a location containing a program code unit suitable for execution by the computer to implement a respective stage of said boot-up procedure, at least one entry pointing toward a location containing a first program code unit residing locally at the computer and at least another entry pointing toward a location containing a second program code unit residing at a site remote from the computer, the first program code unit being associated with a stage of the boot-up procedure that occurs subsequently to the first stage, the first program code unit residing locally at the computer prior to the execution of the first stage;

b) initiating a first stage of said boot-up procedure;

c) processing said policy file to determine a location associated to a given program code unit for effecting a certain stage subsequent to the first stage;

d) accessing said given program code unit at the location determined at step c);

e) processing said given program code unit to cause execution of the certain stage of said boot-up procedure.

17. A method as defined in claim 16, wherein at least one entry in said policy file influences the execution of a stage subsequent to the first stage.

18. A method as defined in claim 17, comprising the step of searching locally for a memory unit.

19. A method as defined in claim 18, comprising the step of searching said memory unit for program code units.

20. A method as defined in claim 17, comprising the step of establishing a data exchange transaction between the computer and the site remote from the computer during execution of either stage of the plurality of stages of said boot-up procedure.

21. A method as defined in claim 20, wherein said policy file includes an entry indicative of a location where a given program code unit resides.

22. A method as defined in claim 21, comprising the step of processing said entry to determine the location where said given program code unit resides.

23. A method as defined in claim 22, comprising the step of attempting to process said given program code unit at the location pointed to by said second entry, upon occurrence of a failure of an attempt to process said given program code unit, attempting to process a program code unit at an another location.

24. A method as defined in claim 23, wherein said second entry points locally at the computer and said another location is a site remote from the computer.

25. A method as defined in claim 23, wherein said second entry points toward a site remote from the computer and said another location is a local site.

26. A method as defined in claim 20, comprising the step of establishing a data exchange transaction between the computer and the site remote from the computer to verify a vintage associated to said policy file, the vintage comprising at least one data element indicative of a time characteristic.

27. A computing apparatus, comprising:

a processor;

a memory holding a program element for directing said processor to execute a multi-stage boot-up procedure of said computing apparatus, said multi-stage boot-up procedure including a first stage and a plurality of successive stages subsequent to the first stage, the implementation of the stages subsequent to the first stage being effected by executing respective program code units, said program element implementing a boot initiation program module for execution by the computing apparatus to implement the first stage of said boot-up procedure;

a policy file in said memory, said policy file having a plurality of entries, each entry pointing to a location containing a program code unit suitable for execution by the computer to implement a respective stage of said multi-stage boot-up procedure, at least one entry pointing toward a location containing a first program code unit residing locally at the computer and at least another entry pointing toward a location containing a second program code unit residing at a site remote from the computer, the first program code unit being associated with a stage of the boot-up procedure that occurs subsequently to the first stage, the first program code unit residing locally at the computer prior to the execution of the first stage.

* * * * *

US006275931B1

(12) **United States Patent**
Narayanaswamy et al.

(10) **Patent No.: US 6,275,931 B1**
(45) **Date of Patent: *Aug. 14, 2001**

(54) METHOD AND APPARATUS FOR UPGRADING FIRMWARE BOOT AND MAIN CODES IN A PROGRAMMABLE MEMORY

(75) Inventors: **Shanthala Narayanaswamy**, Willoughby Hills; **Richard J. Molnar**, Mentor, both of OH (US); **Michael J. Wozniak**, Altamonte Springs, FL (US)

(73) Assignee: **Elsag International N.V.**, Amsterdam (NL)

(*) Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/102,183**

(22) Filed: **Jun. 22, 1998**

(51) **Int. Cl.**^{7} **G06F 9/00**
(52) **U.S. Cl.** 713/2; 711/4
(58) **Field of Search** 713/2, 1; 711/4, 711/1

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,063,496	*	11/1991	Dayan et al. .	
5,214,695	*	5/1993	Arnold et al.	380/4
5,432,927		7/1995	Grote et al.	395/575
5,568,641		10/1996	Nelson et al.	395/700
5,579,522	*	11/1996	Christeson et al.	713/2
5,758,174	*	5/1998	Crump et al.	713/323
5,778,070	*	7/1998	Mattison	380/25
5,805,882	*	9/1998	Cooper et al.	713/2
5,812,390	*	9/1998	Merkin	364/131
5,987,605	*	11/1999	Hill et al.	713/2

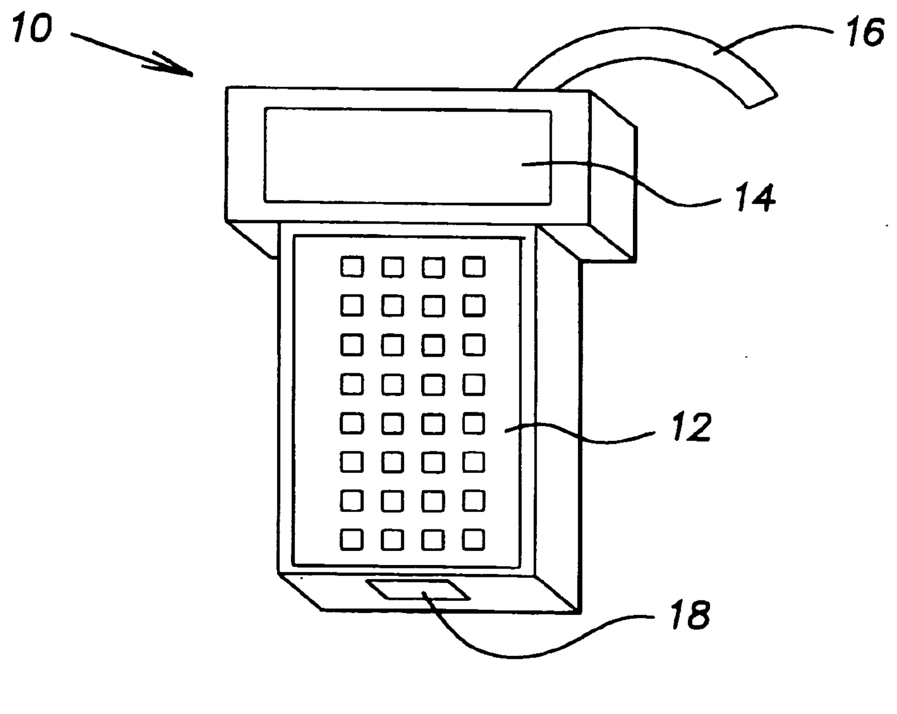
* cited by examiner

Primary Examiner—Robert Beausoleil
Assistant Examiner—Rita A Ziemer
(74) *Attorney, Agent, or Firm*—Michael M. Rickin

(57) **ABSTRACT**

A system for loading upgraded, that is, new boot code and/or main firm ware has a programmable memory that has two boot code regions. One of the regions holds the active boot code while the other region holds the inactive boot code. During a boot code upgrade, the boot code in the inactive region, is under control of the boot code in the active region, replaced with the new boot code. Once the replacement process is verified as having been successful and the vector table in the new boot code is copied to the processor vector table in the memory, the processor can be reset so that the new boot code becomes the active boot code and the previously active boot code becomes the inactive boot code.

34 Claims, 4 Drawing Sheets



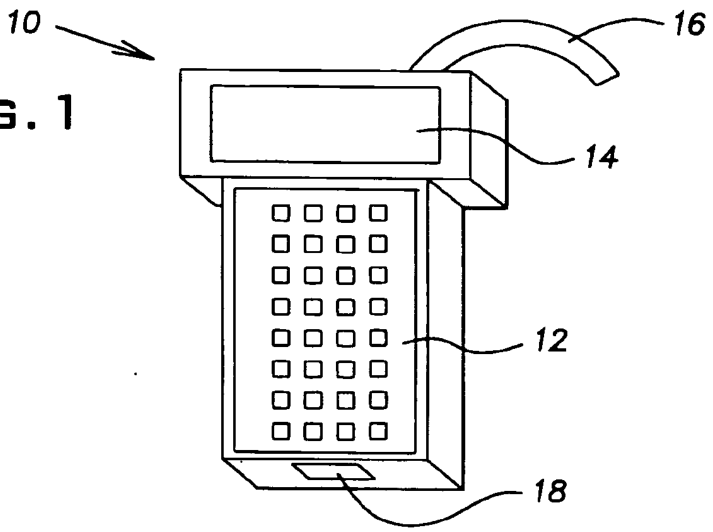

10
16
14
12
18

FIG. 1

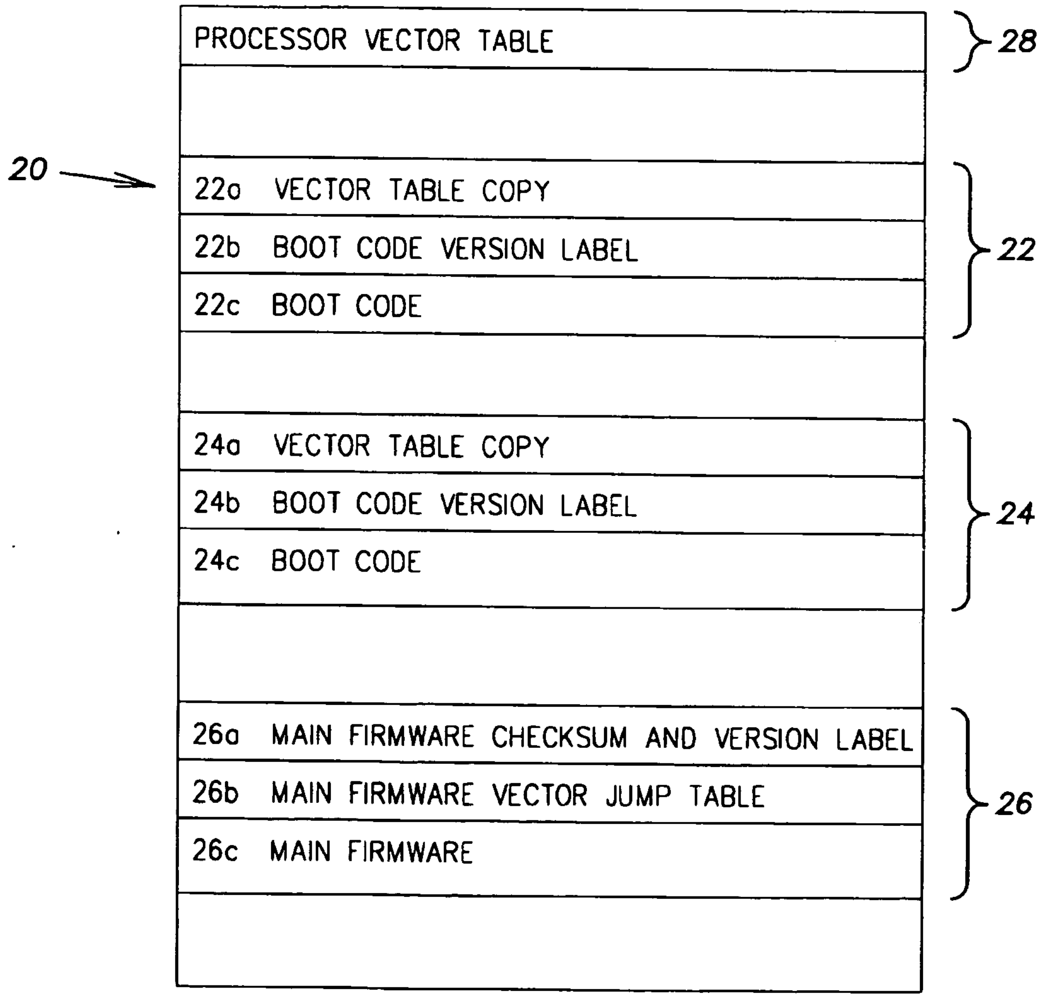

PROCESSOR VECTOR TABLE
28
20
22a VECTOR TABLE COPY
22b BOOT CODE VERSION LABEL
22c BOOT CODE
22
24a VECTOR TABLE COPY
24b BOOT CODE VERSION LABEL
24c BOOT CODE
24
26a MAIN FIRMWARE CHECKSUM AND VERSION LABEL
26b MAIN FIRMWARE VECTOR JUMP TABLE
26c MAIN FIRMWARE
26

FIG. 2

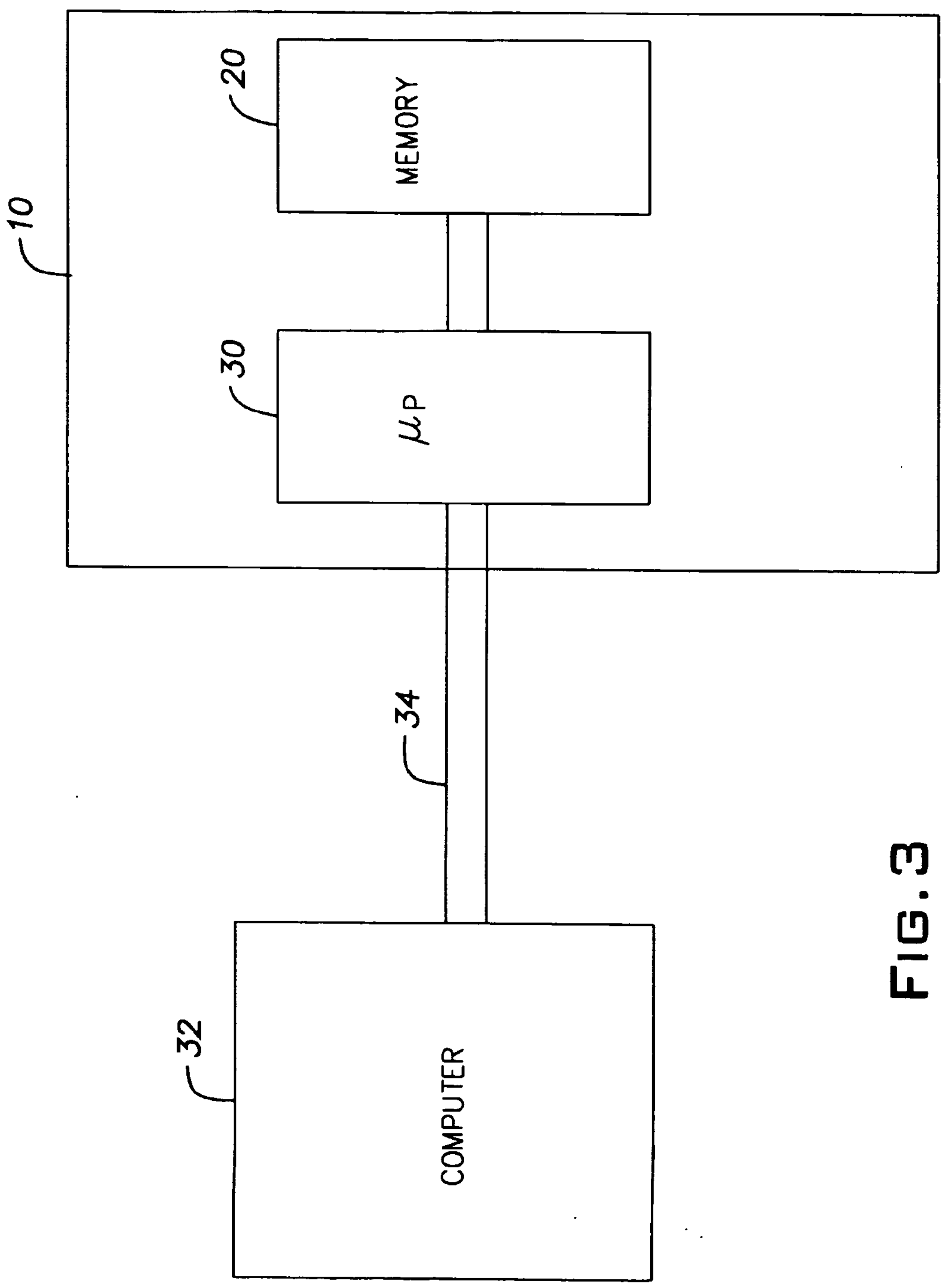


FIG. 3

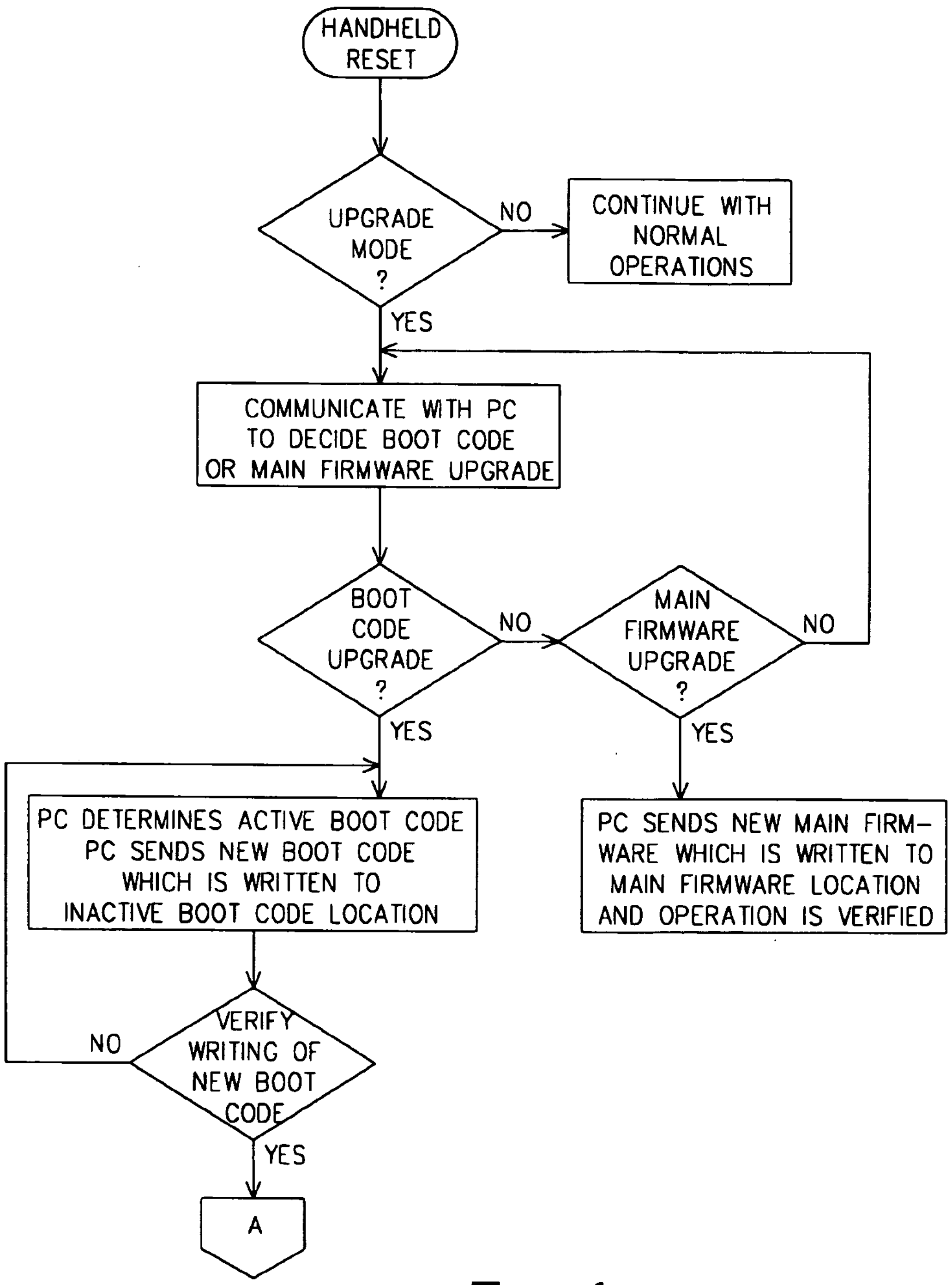

HANDHELD RESET
UPGRADE MODE ?
NO
CONTINUE WITH NORMAL OPERATIONS
YES
COMMUNICATE WITH PC TO DECIDE BOOT CODE OR MAIN FIRMWARE UPGRADE
BOOT CODE UPGRADE ?
NO
MAIN FIRMWARE UPGRADE ?
NO
YES
YES
PC DETERMINES ACTIVE BOOT CODE PC SENDS NEW BOOT CODE WHICH IS WRITTEN TO INACTIVE BOOT CODE LOCATION
PC SENDS NEW MAIN FIRMWARE WHICH IS WRITTEN TO MAIN FIRMWARE LOCATION AND OPERATION IS VERIFIED
NO
VERIFY WRITING OF NEW BOOT CODE
YES
A

FIG. 4A

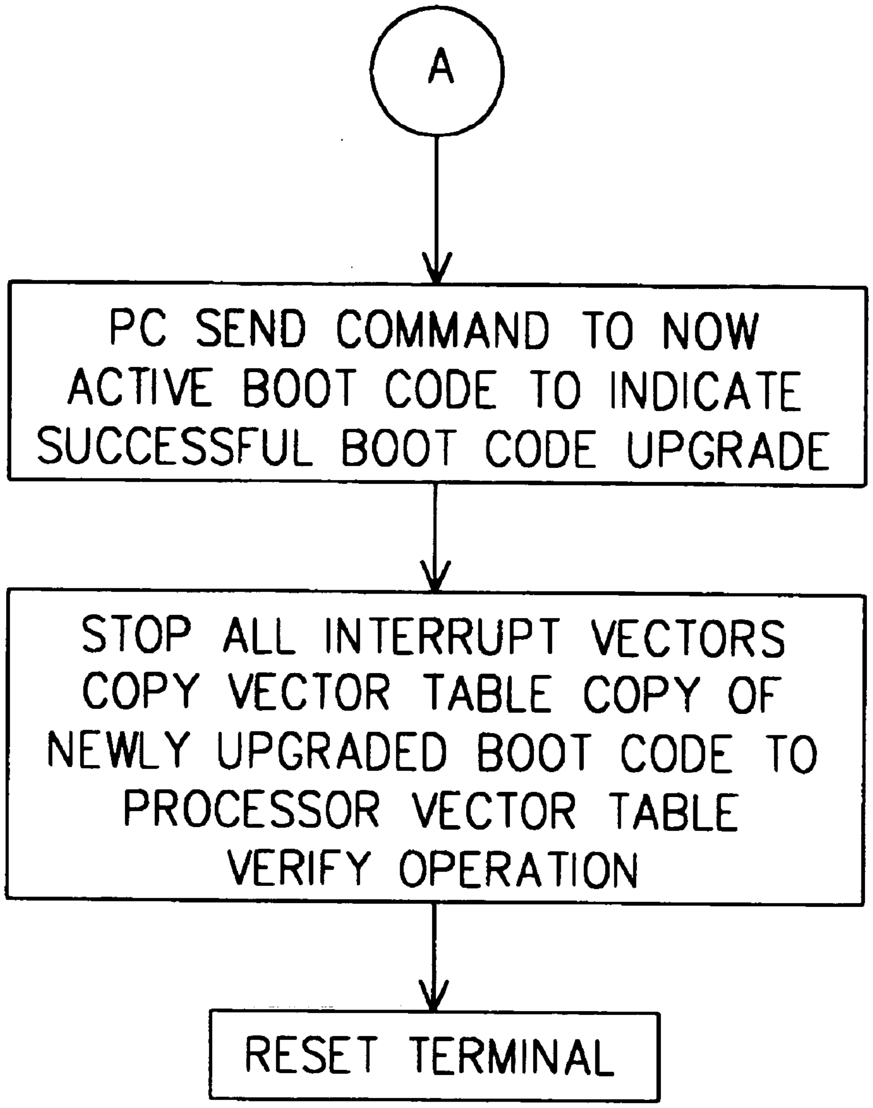

A
PC SEND COMMAND TO NOW ACTIVE BOOT CODE TO INDICATE SUCCESSFUL BOOT CODE UPGRADE
STOP ALL INTERRUPT VECTORS COPY VECTOR TABLE COPY OF NEWLY UPGRADED BOOT CODE TO PROCESSOR VECTOR TABLE VERIFY OPERATION
RESET TERMINAL

FIG. 4B

METHOD AND APPARATUS FOR UPGRADING FIRMWARE BOOT AND MAIN CODES IN A PROGRAMMABLE MEMORY

1. FIELD OF THE INVENTION

This invention relates to the upgrading of either or both of the firmware boot and main codes in a programmable memory.

2. DESCRIPTION OF THE PRIOR ART

Programmable memory and microprocessors are used in many devices. One such device is a remote handheld terminal that is used by technicians in the process industries for process control system configuration, monitoring, tuning and diagnostics. The handheld terminal has firmware therein stored in the programmable memory. The firmware includes the boot code and the main code. The main code is used for the regular operation of the handheld terminal.

Sometimes it is necessary to upgrade the firmware in the remote handheld terminal. A need for upgrading the boot code might arise when the functionality of that code is enhanced. The main firmware will be upgraded whenever the normal functionality of the handheld terminal has to be enhanced or modified. Therefore, the upgrade of firmware in the remote handheld terminal may include an enhancement of the functionality of the terminal. In that instance, the firmware upgrading technique must allow for a flexible size and location remapping of the boot and main firmware codes without any need for hardware changes.

One technique that is now used to upgrade the firmware of a remote handheld terminal is to open the terminal and insert programmable memory with the new firmware therein in place of the programmable memory in the terminal. As can be appreciated, this technique involves both cost and delay in upgrading the terminal to the new firmware as programmable memory must first be programmed with the new firmware and then delivered to the sites where the terminals are used. As can further be appreciated, this technique usually requires an instrument technician or other person with knowledge of electronic circuitry to open and replace the programmable memory, and may result in damage to the handheld terminal during the replacement process.

U.S. Pat. Nos. 5,432,927 and 5,568,641 describe other techniques that may be used to upgrade the boot firmware in a remote handheld terminal. Both of these techniques rely on hardware assisted mapping of the boot code addresses. Therefore, neither of these techniques would allow the size and location of the boot codes in the processor address map of the handheld terminal to be changed without a hardware change. As is described above, such a change is not desirable as it requires that the terminal be opened.

SUMMARY OF THE INVENTION

A system for providing new boot code for a processor.

The system has a writable non-volatile memory. The memory has one region that has active non-write protected boot code therein which has the functionality to allow an instrument containing the memory to communicate with a process control; and another region that has inactive boot non-write protected code therein which is a functional equivalent of the active boot code. The system also has a source of the new boot code; and a processor and associated electronics that is under the control of the active boot code for replacing the inactive boot code with the new boot code from the source.

A system for providing new boot code for a processor. The system has a writable non-volatile memory. The memory has one region that has active non-write protected boot code therein which has the functionality to allow an instrument containing the memory to communicate with a process control; and another region that has inactive non-write protected boot code therein. The system also has a source of the new boot code connected to the processor. The processor operating under control of the active boot code replaces the inactive boot code with the new boot code from the source.

A method of providing new boot code for a processor. The method has a step of providing a writable non-volatile memory having one region having active non-write protected boot code for the processor therein which has the functionality to allow an instrument containing the memory to communicate with a process control and another region having inactive non-write protected boot code for the processor therein which is a functional equivalent of the active boot code. The method also has the steps of connecting a source of new boot code to the processor; transmitting the new boot code from the source to the processor; and writing under control of the active boot code the new boot code to the another region to thereby replace the inactive boot code.

In a device that has a processor a method of providing new boot code for the processor. The method has the step of providing in the device a writable non-volatile memory having one region having active non-write protected boot code for the processor therein which has the functionality to allow an instrument containing the memory to communicate with a process control and another region having inactive non-write protected boot code for the processor therein which is a functional equivalent of the active boot code. The method also has the steps of connecting a source of new boot code to the processor; transmitting the new boot code from the source to the processor; and writing under control of the active boot code the new boot code to the another region to thereby replace the inactive boot code.

A system for providing new boot code for a processor in a device. The system has a writable non-volatile memory in the device. The memory has one region that has active non-write protected boot code therein which has the functionality to allow an instrument containing the memory to communicate with a process control; and another region that has inactive non-write protected boot code therein which is a functional equivalent of the active boot code. The system also has a means for connecting a source of the new boot code to the device; and a means including the processor and under control of the active boot code for replacing the inactive boot code with the new boot code from the source.

A method for upgrading the boot code in an instrument that has a non-volatile memory. The memory has a first boot code block having a vector table and a boot code area and a second boot code block having a vector table and a boot code area. Each of the first and the second boot code areas having boot code in them. The method has the step of determining from a processor vector table stored in the non-volatile memory which of the first and second boot code blocks is then currently active. The method also has the step of writing the upgraded boot code and an associated vector table into the boot code area and vector table area, respectively, of that one of the first and the second boot code blocks which is not then currently active. The method has the further step of determining the successful transfer of the upgraded boot code and the associated vector table to that one of the first and the second boot code blocks which is not then currently active. The method also has the further steps of causing the then currently active boot code to overwrite the processor vector table with the vector table associated with the upgraded boot code; and resetting the instrument upon verification that the overwriting of the processor vector table has occurred so that the upgraded boot code becomes the currently active boot code for the instrument after resetting has occurred.

A method for upgrading the boot code in an instrument having a non-volatile memory. The memory has a first boot code block having a vector table and a boot code area with currently active boot code and a second boot code block having a vector table and a boot code area with currently inactive boot code. The method has the step of writing the upgraded boot code and an associated vector table into the boot code area and vector table area, respectively, of the second boot code block. The method also has the step of determining the successful transfer of the upgraded boot code and the associated vector table to the second boot code block. The method has the further step of causing the then active boot code to overwrite the processor vector table with the associated upgraded vector table. The method also has the further step if resetting the instrument upon verification that the overwriting of the processor vector table has occurred to thereby make the upgraded boot code the active boot code for the instrument after the resetting has occurred.

A method for operating an instrument having a non-volatile memory. The memory has a first boot code block having a vector table and a boot code area with currently active boot code and a second boot code block having a vector table and a boot code area with currently inactive boot code. The method has the step of using the currently active boot code to operate the instrument. The method also has the step of upgrading the currently inactive boot code. The upgrading step has the steps of:

(i) writing the upgraded boot code and an associated vector table into the boot code area and vector table area, respectively, of the second boot code block;

(ii) determining the successful transfer of the upgraded boot code and the associated vector table to the second boot code block;

(iii) causing the active boot code to overwrite the processor vector table with the associated upgraded vector table; and

(iv) resetting the instrument upon verification that the overwriting of the processor vector table has occurred to thereby make the upgraded boot code the active boot code for the instrument after the resetting has occurred.

A system for upgrading boot code in a non-volatile memory of an instrument. The system has a processing device having the upgraded boot code. The instrument is connected to the processing device for receiving the upgraded boot code from the processing device. The instrument non-volatile memory has:

(i) a first boot code block having a vector table and a boot code area with currently active boot code therein;

(ii) a second boot code block having a vector table and a boot code area with currently inactive boot code therein; and

(iii) a processor vector table stored therein.

The processing device causes the instrument to overwrite the currently inactive boot code with the upgraded boot code and the vector table of the second boot code block with a vector table associated with the upgraded boot code, and the currently active boot code to overwrite the processor vector table with the associated upgraded vector table. The processing device also causes the currently active boot code to reset the instrument when the overwriting of the processor

vector table has occurred to thereby make the upgraded boot code the active boot code for the instrument after the resetting has occurred.

A method for upgrading boot code in a non-volatile memory of an instrument. The memory has a first boot code block having a vector table and a boot code area with currently active boot code therein and a second boot code block having a vector table and a boot code area with currently inactive boot code therein. The method has the step of connecting the instrument to a processing device for receiving the upgraded boot code from the processing device. The method also has the step of upgrading the currently inactive boot code. This step has the steps of:

(i) writing the upgraded boot code and an associated vector table into the boot code area and vector table area, respectively, of the second boot code block;

(ii) determining the successful transfer of the upgraded boot code and the associated vector table to the second boot code block;

(iii) causing the active boot code to overwrite a processor vector table stored in the memory with the associated upgraded vector table; and

(iv) resetting the instrument upon verification that the overwriting of the processor vector table has occurred to thereby make the upgraded boot code the active boot code for the instrument after the resetting has occurred.

DESCRIPTION OF THE DRAWING

FIG. 1 shows a handheld terminal.

FIG. 2 shows a simplified layout for the programmable memory in the handheld terminal.

FIG. 3 is a block diagram showing the connection of the handheld terminal to a source of new boot code and/or main firmware.

FIGS. 4A and 4B show a flowchart for the boot code and main firmware upgrade procedures.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

Referring now to FIG. 1, there is shown an example of a remote handheld terminal 10 used in the process control industries. Terminal 10 includes a keypad 12 which includes keys that turn the terminal on and off as well as various keys that allow the technician to configure, monitor and troubleshoot process control field devices. Terminal 10 further includes a display 14 and a cord 16. The cord 16 has clip leads (not shown) which are used to clip onto the signal wires of the field devices.

Handheld terminal 10 further includes an RS232 port 18 which may, be for example, be located in the bottom of terminal 10 as is shown in FIG. 1. Port 18 allows the handheld terminal 10 to be connected by a suitable cable to the serial port of a personal computer (PC) 32 [see FIG. 3] so that in accordance with the present invention the boot and/or main firmware in the terminal can be upgraded.

Internal to the terminal 10 is a programmable memory 20, a simplified layout for which is shown in FIG. 2, and a microprocessor and associated electronics 30 (see FIG. 3). Memory 20 may for example be a Flash electrically erasable programmable read-only memory. As is shown in FIG. 2, the memory 20 is partitioned into functional software units where the boot code and main firmware reside. Specifically, in memory 20 there are first and second boot code units 22, 24, main firmware unit 26, and process vector table 28. As will be described in more detail below, the first boot code and second boot code are not simultaneously active. When the first boot code is active the second boot code is inactive and vice versa.

Each of the boot code units 22, 24 include a copy of the vector table 22*a*, 24*a* at the top of each of units 22, 24. Directly below the vector table, each unit includes a version label 22*b*, 24*b* and directly below the version label each unit includes the boot code 22*c*, 24*c*. The main firmware unit includes at its top a checksum and version label 26*a*. Directly below the checksum and version label, the main firmware table includes a vector jump table 26*b*.

The main firmware 26*c* is directly below the vector jump table.

The upgrading of the boot code will first be described below followed by a description of the upgrading of the main firmware. When the boot code and/or main firmware in terminal 10 is to be upgraded, the RS232 port 18 of terminal 10 is, as is shown in the block diagram of FIG. 3, connected to the serial port 36 of the PC 32 by cable 34. PC 32 includes the application software that communicates with the terminal 10 during the upgrade.

When terminal 10 is operating, either boot code 1 or boot code 2, but not both, is active. The microprocessor 30 operates under control of the then active boot code to perform the upgrade of the boot code and/or main firmware. In the boot code upgrade, the inactive boot code is replaced with upgraded, that is, new boot code from PC 32. When terminal 10 and thus microprocessor 30 is reset, the new boot code becomes the active boot code. In the main firmware upgrade, the main firmware then in terminal 10 is replaced with new main firmware from the PC 32.

In both upgrades the files for the upgraded boot code firmware and the upgraded main firmware are made available to a registered user of the handheld terminal 10 on an Internet web site. The registered user can log on to the web site and download the upgrade file(s) from the web site onto PC 32. The boot code and main firmware addresses are encrypted in the associated upgrade file so that PC 32 can control the addresses in memory 20 into which the upgraded boot code and/or main firmware are written. The files for the upgraded firmware that are posted on the web site may be encrypted for purposes of security and also contain the version number and checksum information.

Boot Code Upgrade

In the procedure for upgrading of the boot code, the new, that is, upgraded, boot code replaces the boot code in the inactive boot code block of terminal 10. For example, if boot code block 22 is active during the boot code upgrade then the new boot code will be written into boot code block 24. Upon the resetting of terminal 10 and thus microprocessor 30, the new boot code will become the active boot code of the handheld terminal 10.

The specific procedure for upgrading the boot code including swapping the upgraded boot code for the non-upgraded boot code is as follows:

1. The PC application software sends a command to the terminal 10 to read the processor vector table 28 to thereby determine which one of the boot code blocks 22, 24 is currently active in the terminal. For purposes of explanation it will be presumed hereinafter that boot code 22*c* is active in terminal 10.
2. The PC then requests the handheld active boot code 22*c* to start tracking a checksum on the data, that is the new boot code, being written into the inactive block 24. The new code is then sent to the inactive boot code block 24.

The new boot code can be written to a start address that is different than the start address of the inactive boot code. This flexibility allows boot code block 24 to be moved to accommodate an increase in the size of the new boot code now being written to block 24 or an increase in the size of boot code block 22 that is planned to appear in the next upgrade. It should be appreciated that the new boot code should not overwrite any part of the active boot code. The relocation of block 24 is taken into account when vector table copy 24*a* is written into the processor vector table 28 at the end of the upgrade procedure.

3. After the PC upgrades the inactive boot code block 24, the PC confirms the checksum of the newly loaded upgraded boot code 24.
4. Upon confirmation of the checksum the PC sends a command to terminal 10 to indicate that the new boot code firmware was successfully transferred to boot code block 24. This command is also a request to the active boot code 22*c* to turn off all interrupt activity and copy the new vector table copy 24*a* transmitted from the PC into the processor vector table 28 and reset the handheld terminal 10. The vector table copy 24*a* of the new boot code 24*c* has to be transferred to the processor vector table 28 of memory 20 to make the new boot code 24*c* the active boot code of the terminal 10.
5. In response to the command from the PC to transfer the new vector table copy 24*a* into the processor vector table 28, the terminal 10 shuts off all of its interrupts in order to make sure that none of the interrupt vectors are used. The terminal then overwrites the processor vector table 28 with the vector table copy 24*a* of the newly upgraded boot code 24*c*.
6. Upon verification by the now active boot code 22*c* that the overwrite of processor vector table 28 has occurred, the boot code 22*c* resets the terminal 10. Upon terminal 10 reset, the upgraded boot code 24*c* becomes the active boot code of the terminal.

It should be appreciated that at the end of the successful upgrade of boot code block 24 described above, boot code 24*c* of block 24, that is the new boot code, is the active boot code for the handheld terminal 10 and boot code 22*c* of block 22 is inactive as that is the old, that is, not upgraded, boot code. The next time the boot code is to be upgraded, it will be boot code 22*c* in inactive block 22 that is upgraded. At the successful end of that upgrade, the new boot code of boot code 22*c* of block 22 will become the active boot code for the terminal 10 and the old boot code of boot code 24*c* of block 24 will become the inactive boot code. Therefore, each successful upgrade of the boot code causes the previously inactive boot code block to become the active boot code and the previously active boot code block to become the inactive boot code block.

Main Firmware Upgrade

The specific procedure for upgrading the main firmware is as follows:

1. The PC application program sends commands to the terminal 10 to prepare the terminal for the upgrade. These commands request the active boot code to start checking a checksum on the data, for, the new, that is, upgraded main firmware to be written into the memory 20.
2. The PC transmits the new main firmware to terminal 10. The new main firmware can be written to a start address which is different than the start address of the present main firmware. This flexibility allows the main firmware block 26 to be moved to accommodate an

increase in size of the main firmware now being written or a planned increase in the size of the next upgrade of the main firmware or the boot code.

3. The PC confirms the checksum check at the end of the transfer to verify that the transfer was successful.

4. Upon verification of a successful transfer, the PC will send a reset command to the terminal **10** so that the terminal can start running the new firmware.

The interrupt vectors in the main firmware are double indexed through a fixed table. The processor vector table **28** will point to a location in the main firmware vector jump table **26*b*** which in turn points to a location in the main firmware **26*c***. Therefore, even if the main firmware vector jump table **26*b*** is changed by the main firmware upgrade the double indexing avoids the need to upgrade the processor vector table **28** at the end of the main firmware upgrade.

FIGS. **4A** and **4B** show a flowchart for the boot code and main firmware upgrade procedures described above.

As was described above, the boot code and main firmware addresses are encrypted in the associated upgrade file so that the PC can control the addresses in memory **20** into which the upgraded boot code and/or main firmware are written. Therefore, the size and starting address of the boot code blocks **22, 24** and the main firmware block **26** in memory **20** are not fixed and can be changed through the upgrade procedure. The only limitations on increasing the size of blocks **22, 24** and **26** are the size of memory **20** and the size of an adjacent block during the upgrade of blocks **22, 24** and **26**. For example, an increase in the size of main firmware block **26** through the upgrade procedure described above is limited by adjacent block **24**. It should be appreciated that a series of upgrades can result in the increase of the size of block **26** by moving and/or shrinking blocks **22** and **24**.

A boot code or main firmware upgrade procedure may fail prior to completion for any one of a number of reasons including a power failure or a break in the cable **34** connecting port **18** to the PC. Even if the procedure were to fail prior to completion, the terminal **10** is still operable. If the main firmware upgrade fails prior to completion the operator can start terminal **10** again using the boot code firmware contained in memory **20** and reinitiate the main firmware upgrade. During a boot code firmware upgrade the boot code being upgraded is in the inactive boot code block. Therefore a failure in the upgrade prior to completion is not a problem as the terminal can still operate using the active boot code.

The flexibility described above in writing new boot code or main firmware is also applicable where the new code is the same size as or less even than the code it is replacing.

It is to be understood that the description of the preferred embodiment(s) is (are) intended to be only illustrative, rather than exhaustive, of the present invention. Those of ordinary skill will be able to make certain additions, deletions, and/or modifications to the embodiment(s) of the disclosed subject matter without departing from the spirit of the invention or its scope, as defined by the appended claims.

What is claimed is:

1. A system for providing new boot code for a processor comprising:

a. a writable non-volatile memory comprising:
   i. one region having active non-write protected boot code therein, said boot code having the functionality to allow an instrument containing said memory to communicate with a process control system; and
   ii. another region having therein an inactive non-write protected boot code which is a functional equivalent of said active boot code;

b. a source of said new boot code; and

c. means including said processor and under control of said active boot code for replacing said inactive boot code with said new boot code from said source.

2. The system of claim **1** wherein said memory further comprises a processor vector table and said new boot code includes a vector table, said replacing means causing after said new boot code has replaced said inactive boot code said new boot code vector table to be copied to said memory processor vector table so that said new boot code can become said active boot code upon resetting of said processor.

3. A system for providing new boot code for a processor comprising:

a. a writable non-volatile memory comprising:
   i. one region having non-write protected active boot code therein, said boot code having the functionality to allow an instrument containing said memory to communicate with a process control system; and
   ii. another region having therein an inactive non-write protected boot code which is a functional equivalent of said active boot code; and

b. a source of said new boot code connected to said processor;

said processor operating under control of said active boot code for replacing said inactive boot code with said new boot code from said source.

4. The system of claim **3** wherein said memory further comprises a processor vector table and said new boot code includes a vector table, said processor causing after said new boot code has replaced said inactive boot code said new boot code vector table to be copied to said memory processor vector table so that said new boot code can become said active boot code upon resetting of said processor.

5. A method of providing new boot code for a processor, comprising the steps of:

a. providing a writable non-volatile memory having one region having active non-write protected boot code for said processor therein, said boot code having the functionality to allow an instrument containing said memory to communicate with a process control system and another region having therein an inactive non-write protected boot code which is a functional equivalent of said active boot code;

b. connecting a source of new boot code to said processor;

c. transmitting said new boot code from said source to said processor; and

d. writing under control of said active boot code said new boot code to said another region to thereby replace said inactive boot code.

6. The method of claim **5** further including after said writing step the step of verifying that said new boot code is written to said another region.

7. The method of claim **6** wherein said memory also has a processor vector table and said new boot code includes a vector table and said method further includes after said verifying step the step of copying said new boot code vector table to said memory processor vector table.

8. The method of claim **7** further including after said copying step the step of resetting said processor so that said new boot code becomes said active boot code.

9. In a device having a processor a method of providing new boot code for said processor, comprising the steps of:

a. providing in said device a writable non-volatile memory having one region having active non-write protected boot code for said processor therein, said boot code having the functionality to allow an instrument containing said memory to communicate with a process control system and another region having therein an inactive non-write protected boot code which is a functional equivalent of said active boot code;

b. connecting a source of new boot code to said processor;

c. transmitting said new boot code from said source to said processor; and

d. writing under control of said active boot code said new boot code to said another region to thereby replace said inactive boot code.

10. The method of claim 9 wherein said new boot code source is external to said device.

11. The method of claim 9 further including after said writing step the step of verifying that said new boot code is written to said another region.

12. The method of claim 11 wherein said memory also has a processor vector table and said new boot code includes a vector table and said method further includes after said verifying step the step of copying said new boot code vector table to said memory processor vector table.

13. The method of claim 12 further including after said copying step the step of resetting said processor so that said new boot code becomes said active boot code.

14. A system for providing new boot code for a processor in a device, comprising:

a. a writable non-volatile memory in said device, said memory comprising:

i. one region having active non-write protected boot code therein, said boot code having the functionality to allow an instrument containing said memory to communicate with a process control system; and

ii. another region having therein an inactive non-write protected boot code which is a functional equivalent of said active boot code;

b. means for connecting a source of said new boot code to said device; and

c. means including said processor and under control of said active boot code for replacing said inactive boot code with said new boot code from said source.

15. The system of claim 14 wherein said device has a housing and said processor, said replacing means, and said memory are inside said housing.

16. The system of claim 15 wherein said new boot code source is outside of said housing.

17. The system of claim 15 wherein said source includes a connector and said means for connecting said source to said device comprises a connector on said housing and a cable compatible with said source connector and said housing connector.

18. A method for upgrading the boot code in an instrument having a non-volatile memory having a first boot code block having a vector table and a boot code area therein, and a second boot code block having a vector table and a boot code area therein, each of said first and said second boot code areas having non-write protected boot code therein, said method comprising the steps of:

(a) determining from a processor vector table stored in said non-volatile memory which of said first and said second boot code blocks is then currently active, said boot code in said first boot code area having the functionality to allow said instrument to communicate with a process control system;

(b) writing said upgraded boot code and an associated vector table into said boot code area and vector table area, respectively, of that one of said first and said second boot code blocks which is not then currently active;

(c) determining the successful transfer of said upgraded boot code and said associated vector table to that one of said first and said second boot code blocks which is not then currently active;

(d) causing said then currently active boot code to overwrite said processor vector table with said vector table associated with said upgraded boot code; and

(e) resetting said instrument upon verification that said overwriting of said processor vector table has occurred so that said upgraded boot code becomes the currently active boot code for said instrument after resetting has occurred.

19. The method of claim 18 wherein in said writing step said upgraded boot code is written to a start address in said boot code area of that one of said first and said second boot code blocks which is not then currently active which is different than the start address of the boot code in said boot code area of that one of said first and said second boot code blocks which is then currently active.

20. The method of claim 18 wherein said boot code in said first boot code block boot code area and said boot code in said second boot code block second area are functional equivalents of each other.

21. The method of claim 18 wherein said step of determining which of said first and said second boot code blocks is then currently active is preceded by the step of connecting said instrument to a processing device, said processing device obtaining said upgraded boot code from a remote source.

22. A method for upgrading the boot code in an instrument having a non-volatile memory having a first boot code block having a vector table and a boot code area with currently active non-write protected boot code therein, and a second boot code block having a vector table and a boot code area with currently inactive non-write protected boot code therein, said currently active boot code having the functionality to allow said instrument to communicate with a process control system, said method comprising the steps of:

(a) writing said upgraded boot code and an associated vector table into said boot code area and vector table area, respectively, of said second boot code block;

(b) determining the successful transfer of said upgraded boot code and said associated vector table to said second boot code block;

(c) causing said then active boot code to overwrite a processor vector table stored in said memory with said associated upgraded vector table; and

(d) resetting said instrument upon verification that said overwriting of said processor vector table has occurred to thereby make the upgraded boot code said active boot code for said instrument after said resetting has occurred.

23. The method of claim 22 wherein in said writing step said upgraded boot code is written to a start address in said boot code area of said second boot code block which is different than the start address of said currently active boot code in said boot code area of said first block.

24. The method of claim 22 wherein said currently active boot code and said currently inactive boot code are functional equivalents of each other.

25. The method of claim 22 wherein said writing step is preceded by the step of connecting said instrument to a processing device, said processing device obtaining said upgraded boot code from a remote source.

26. A method for operating an instrument having a non-volatile memory, said memory having a first boot code block

having a vector table and a boot code area with currently active non-write protected boot code therein, and a second boot code block having a vector table and a boot code area with currently inactive non-write protected boot code therein, said method comprising the steps of:

(a) using said currently active boot code to operate said instrument, said currently active boot code having the functionality to allow said instrument to communicate with a process control system; and

(b) upgrading said currently inactive boot code comprising the steps of:

(i) writing said upgraded boot code and an associated vector table into said boot code area and vector table area, respectively, of said second boot code block;

(ii) determining the successful transfer of said upgraded boot code and said associated vector table to said second boot code block;

(iii) causing said active boot code to overwrite a processor vector table stored in said memory with said associated upgraded vector table; and

(iv) resetting said instrument upon verification that said overwriting of said processor vector table has occurred to thereby make said upgraded boot code the active boot code for said instrument after said resetting has occurred.

27. The method of claim **26** wherein in said writing step said upgraded boot code is written to a start address in said boot code area of said second boot code block which is different than the start address of said currently active boot code in said boot code area of said first block.

28. The method of claim **26** wherein said currently active boot code and said currently inactive boot code are functional equivalents of each other.

29. The method of claim **26** wherein said writing step is preceded by the step of connecting said instrument to a processing device, said processing device obtaining said upgrade boot code from a remote source.

30. The method of claim **26** wherein said step of upgrading said inactive boot code terminates prior to the completion of said writing step and said instrument continues to operate using only said currently active boot code.

31. A system for upgrading boot code in a nonvolatile memory of an instrument comprising:

(a) a processing device having said upgraded boot code, said instrument connected to said processing device for receiving said upgraded boot code from said processing device;

said instrument non-volatile memory comprising:

(i) a first boot code block having a vector table and a boot code area with currently active non-write protected boot code therein, said currently active boot code having the functionality to allow said instrument to communicate with a process control system;

(ii) a second boot code block having a vector table and a boot code area with currently inactive non-write protected boot code therein; and

(iii) a processor vector table stored therein;

said processing device causing said instrument to overwrite said currently inactive boot code with said upgraded boot code and said vector table of said second boot code block with a vector table associated with said upgraded boot code, and said currently active boot code to overwrite said processor vector table with said associated upgraded vector table;

said currently active boot code resetting said instrument when said overwriting of said processor vector table has occurred to thereby make said upgraded boot code the active boot code for said instrument after said resetting has occurred.

32. The system of claim **31** wherein said processing device receives said upgraded boot code from a remote source.

33. A method for upgrading boot code in a nonvolatile memory of an instrument, said memory having a first boot code block having a vector table and a boot code area with currently active non-write protected boot code therein, said currently active boot code having the functionality to allow said instrument to communicate with a process control system and a second boot code block having a vector table and a boot code area with currently inactive non-write protected boot code therein, said method comprising the steps of:

(a) connecting said instrument to a processing device for receiving said upgraded boot code from said processing device;

(b) upgrading said currently inactive boot code, said currently inactive boot code a functional equivalent of said currently active boot code, said upgrading comprising the steps of:

(i) writing said upgraded boot code and an associated vector table into said boot code area and vector table area, respectively, of said second boot code block;

(ii) determining the successful transfer of said upgraded boot code and said associated vector table to said second boot code block;

(iii) causing said active boot code to overwrite a processor vector table stored in said memory with said associated upgraded vector table; and

(iv) resetting said instrument upon verification that said overwriting of said processor vector table has occurred to thereby make said upgraded boot code the active boot code for said instrument after said resetting has occurred.

34. The method of claim **33** further comprising before said step of connecting said instrument to said processing device the step of connecting said processing device to a remote source for receiving said upgraded boot code from said remote source.

* * * * *